Contents

1 What is AI

Areas with ai

- Healthcare
- Customer Service
- Transport
- Manufacturing
- Games
- Smart homes

To define intelligence, we think about how the mind arises from the brain, and what intelligent behaviors they cause. Some intelligent behaviors are

- Collect and interpret information from the environment
- Conversation and speech
- Logical reasoning (cause and effect)
- Navigation and mobility
- Humor
- Morality

The four methods of defining AI are

- Thinking like a human, self-awareness, emotional but maybe not rational. Can find how we think by experimentation
- Thinking rationally, logical reasoning
- Acting like a human, abstract problem solving, learning and memory, language, intuition, consciousness, emotions, adapting (can be tested using the turing test)
- Acting rationally, performs actions that will achieve their goals. It might not have perfect knowledge, but it should do the most rational action.

The turing test is a test by Alan Turing in 1950, it involves a human conversing with a computer (that they don't know) via teletype. The aim is for the computer to imitate a human well enough to fool them. The Total turing test allows physical objects to be passed to the machine as well as characters. Eliza was an early chatbot that attempted to pass the test. The problem is that the turing test is not reproducible, constructive (only evaluation), or able to be benchmarked using maths.

This course views intelligence as agents acting rationally.

State-of-the-art problems that ai can solve

- Playing table tennis
- Driving along roads
- Playing board games
- Prove a new maths theorem

- Write an intentionally funny story
- Translate languages in real time

2 Agents

Goal is to relate the complexity of agents to the difficulty of the tasks that they need to perform.

To characterize an agent via a model, we consider

- Percepts/observations of the environments by sensors
- Actions that can affect the environment, made by actuators
- Environment that the agent is in
- Performance measures of the desirability of the environment states.

We can model an agent as a function, mapping from percept sequences to actions, so we can evaluate them mathematically. The ideal rational agent would pick actions that maximize its expected performance measure based on the percept and the model function. Note that rational does not imply omniscient or successful.

Types of agents are

- Simple reflex agents, the actions are decided based on the current percepts and a set of rules (if else). Assumes that the percepts describes everything about the environment. Advantage is that it is stateless and fast. (The sensor module only takes the current percepts to derive the world state)
- Model based reflex agents, similar to simple reflex agents, but it assumes that the percepts do not observe all about the world, and uses past states, how the world evolves over time, and how its actions can affect the world to determine the environment state. (The sensor module contain world state transitions, and use the current percepts and state transitions to derive the current state)
- Goal based agents, like a model-based reflex agent, but also considers the impact of each action and chooses the action that achieves some goal (choose move that wins a chess game)
- Utility based agents, like a goal-based agent, but substitute the goal with a utility function on the potential forward state, and use that to determine the action to do (choose move that maximize winning prob in chess)

Basically

- Simple reflex has no memory and forward-thinking
- Model based reflex agents has memory and no forward-thinking
- Goal based agents has memory and forward-thinking
- Utility based agents has memory, forward-thinking, and can measure utility of states. More general than goal based agents since utility can blend multiple goals and can differentiate between actions that achieve the same goal but with different preferences.

Examples

- Oven, car braking system

- Obstacle detection system, positioning system
- Chess engine, self-driving agent
- Stock market bot, scheduling system

The types and factors of environments

- Observable, percepts contains all relevant information
- Deterministic, the current state unique determines the next
- Episodic, only the current percept is relevant, short term actions does not have long term consequences, markov chain
- Static, environment doesn't change when agent is thinking
- Discrete, finite number of percepts/actions.

The opposites are: partially-observable, stochastic, sequencial, dynamic, continuous. The real world is all of the opposites.

3 Problem Formulation

An offline problem solving general agent looks like

```
function solve(percept):
state = update_state(state, percept)
if empty(action_sequence):
g = formulate_goal(state)
problem = formulate_problem()
action_sequence = search(state, problem)

action = recommendation(action_sequence, state)
action_sequence = remainder(sequence, s2tate)
return action
```

Essentially, it maintains a state and action sequence. Initially, it will compute the goal, problem, and the action sequence from the current state. For every state after, it will get the action from the initially computed action sequence and update the state. This is offline because it assumes that we have complete knowledge of the problem and the solution (can get the action sequence from an initial state). Online versions will compute the action sequence at every new state.

To use this general solving function, we need to do

- Formulate goal
- Formulate problem: get states and actions/operators
- Find solutions (action sequences)

A single-state problem is a specific type of problem where we know which state we are in (completely observable currently, so only one unique state we can be in). To formulate a single-state problem for analysis, consider

- Initial state

- Actions, or successor function
- Goal test. Explicit is an equality test against a single state, implicit is a set inclusion against a set of states.
- Path cost, the additive cost of all the actions taken so far
- And the solution will be a sequence of actions (operators) from the initial state to the goal state.

To select a state space (for single state problem formulation), we need to abstract the real world states for this specific problem (because the real world is abstract).

- Abstract state will be a set of real world state
- Abstract action is a combination of real world actions. To guarantee that this is realized (implementable), we just need to ensure for all (state1 to state2), any real state in state1 can have an action to some real state in state2.
- Abstract solution is a set of real paths that are solutions in the real world

Importantly, the abstract states and actions must be simpler than the real world states and actions.

3.1 Searching Agent

The basic idea of an offline search will simulate the exploration of the state space by generating successors from explored states. The function requires a strategy function to select the next node to explore

```
function search(problem, strategy):
initial_search_tree(problem)
loop:
if no_candidates:
return failure

node = choose_leaf_node(strategy)
if node_contains_goal(node):
return solution

new_nodes = expand_node(node)
add_new_nodes_to_tree(new_nodes)
```

The actual implementation would be

```
function search(problem, strategy):
nodes = queue
loop:
if empty(nodes):
return failure

node = front(nodes)
if goal_test(node):
return node
```

```
strategy(nodes, expand(node, operators(problem)))
```

where we use a queue and a queueing strategy.

The difference between states and nodes

- States represent an abstract real world physical configuration
- Node is a data structure in the search tree. It contains a state, depth, path cost, parent, child, etc
- There is an expand function that creates new nodes from an existing node, and it uses the action function to create the corresponding child states.

A strategy is defined by the order of nodes to visit/expand (via the queueing strategy). They are evaluated against

- completeness, does it always find a solution if it exists
- time complexity on nodes generated
- space complexity, max nodes in memory
- optimality, does it find the least cost solution

For the complexities, we use the parameters

- b is the maximum branching factor
- m is the maximum depth there are
- d is the depth of the least cost solution

4 Uninformed Search

Uninformed search algorithms use only the information from the problem (state transitions) without any heuristics.

Breadth first search will expand the shallowest unexpanded node, namely, put the successor at the end of the queue. The properties are

- Complete
- Time complexity $O(b^d)$
- Space complexity $O(b^d)$
- Not optimal.

Uniform cost search will expand the least-cost unexpanded node (Dijkstra), namely, insert into the queue via increasing path cost. The properties are

- Complete, if path costs are non-zero
- The time complexity is the number of nodes where the path cost is less than the path cost of the optimal solution
- Space complexity the space as time complexity
- It is optimal

Depth first search will expand the deepest unexpanded node. The queueing function inserts the successor at front of queue. It can reach an infinite loop if the search space is cyclic. We will either need a domain that is finite and acyclic, or to keep a list of visited states and ignore those.

- Not complete if $m = \infty$ or cyclic search space. Complete if otherwise
- Time complexity $O(b^m)$. Bad if m is much larger than d. But if solutions are dense in graph, could be faster than bfs.
- Space complexity $O(bm)$
- Not optimal

Iterative deepening search repeatedly does depth limited search with limit increasing when there are no solutions. THe properties are

- Complete
- Time complexity $O(b^m)$
- Space complexity is $O(bd)$
- Not optimal (but can be with action ordering)

Bidirectional search searches simultaneously forward from the start node, and backwards from the goal state, and stop if the two searches meet in the middle to the same node. Using BFS, its properties are

- Complete
- Time $O(b^{d/2})$
- Space $O(b^{d/2})$
- Optimal if using the right strategy

This meet in the middle search will only work if the action is reversible.

5 Informed Search

Informed search uses a heuristic to improve the search algorithm that contains domain specific knowledge about the environment.

5.1 Best First Search

- Uses an evaluation function on each state that estimate the desirability of the node.
- Best-First-Search will expand the most desirable unexpanded node.
- The Queuing function will have the nodes in decreasing levels of desirability

Define the heuristic function $h(n)$ for node n. It estimates the cost from n to the goal. For shortest path, a reasonable heuristic is the euclidean distance between the node state and the goal state.

Greedy search will only consider the heuristic of nodes in the queue. This means that it will expand the closest node to the target that is unexpanded.

- Complete if finite state space and repeated state checking. But can stuck in loops otherwise (entered into a deadend loop).
- Time complexity, $O(b^m)$ but a good heuristic can lead to large improvements.
- Space is $O(b^m)$ since we keep all nodes in memory
- It is not optimal

A* will consider the sum of the heuristic and the path cost of nodes in the queue, namely $f(n) = g(n) + h(n)$ where $f(n)$ is total cost and $g(n)$ is the path cost. Intuitively, $f(n)$ is the estimated total cost from n to goal.

- The idea is to avoid expanding paths that's already expensive due to the path cost
- The heuristic must be admissible, namely that $h(n) \leq h^*(n)$ where $h^*(n)$ is the true cost. So the heuristic must have a cost less than or equal to the true cost (never overestimates the cost)

Theorem, A* is optimal. Optimal in both: the path found is the shortest path, no other algorithms exist that can find the solution in fewer nodes. To prove the optimality, consider the queue with both an suboptimal goal, and we should derive that the suboptimal goal will never get expanded since

$$\begin{aligned} f(sub) &= g(sub) \\ &> g(goal) \\ &> f(goal) \end{aligned}$$

This sub-optimal node is never expanded since A* expands node in increasing $f(n)$.

Also note that A* expands nodes in order of increasing $f(n)$ values. This is because it generates f-contours of nodes where the ith contour has nodes where $f(n) = f_i$, and $f_i > f_{i-1}$.

Properties of A*

- Complete unless infinity many nodes where $f < f(goal)$
- Time is exponential in (relative error of h $\times$ length of solution). Smaller the error the lower the time complexity
- Space complexity, same as time since all nodes are in memory
- Optimal, yes

5.2 Admissible Heuristic

Note that an admissible heuristic must have the property that it never overestimates the true costs to the goal state.

If $h_2(n) \geq h_1(n)$ for all n, and both are admissible, we say that h_2 dominates h_1. A theorem is that a dominant heuristic will always perform better in A* (this is because A* searches all nodes with $f < f(goal)$, since $h(goal) = 0$, and increasing h will increase all other f, it must search less than or equal to the number of nodes)

We can derive the admissible heuristic from the exact solution cost of a reduced version of the problem. The reduced version should have the cost to the goal state be reduced.

5.3 Iterative Improvement algorithm

If the path to a solution state doesn't matter, and the goal state itself is a solution, we can use a simpler algorithm.

- The state space is a set of all configurations
- Find an optimal configuration/state, or finding a configuration that satisfies some constraint

We can solve this using iterative improvement, where we take a single state, and iterative update it to a better solution. It will take constant space, and can be done online (searching by doing an irl move) and offline.

Hill climbing

```
function hillclimb
current = state
while true
next_state = highest_eval_neighbor(current)
if evaluation(next_state) < evaluation(current)
return current
end
current = next_state
end
end
```

We can also do gradient descent by moving to the neighbor with the highest evaluation.

The problem with hill climbing is that we can get stuck in a local maxima.

6 Adversarial Search

Games as problem solving

- Opponents are unpredictable, we need a contingency plan (decision trees on future actions) as a solution instead of a sequence of actions
- There are time limits, unlikely to find goal, must approximate
- Use methods like: perfect play algorithm minimax, finite horizon and approximate evaluation, pruning

Types of games, two dimensions: perfect information, deterministic. We can view imperfect information as chance like selecting a card from a deck

- Perfect information, Deterministic, Chess
- Perfect information, Chance, Monopoly
- Imperfect information, Deterministic, Battleship
- Imperfect information, Chance, Poker

To represent a strategic two-player game (need to think ahead) as a search problem, we need

- The initial state

- The actions
- Terminal test when game ends
- Utility function when game ends. In zero-sum game with two players (zero total utility), the players utility for a state are equal and opposite

6.1 Minimax

Perfect play for deterministic, perfect information games. Idea is to choose move that leads to a position with the highest minimax value (assuming best play from opponent).

- Each depth is either a Max level or a Min level. Max level happens if the initial player is moving, Min level is the other player.
- The score for a node is either the state utility if it is terminal, or the respective Min or Max of its child node scores depending on its level.
- Note that the utility for a state is always against the initial player
- A move by a player is called a ply

Pseudocode

```
function minimax_decision(game)
for action in operators(game)
value = minimax_value(apply(action, game))
end

return action with highest value
end

function minimax_value(game)
if terminal(state(game))
return utility(game)
end

for action in operators(game)
value = minimax_value(apply(action, game))
end

if max is moving
return highest value
else if min is moving
return minimum value
end
end
```

Properties

- Complete, since all reachable states will be visited, if tree is finite (draws for like chess)
- Optimal, assume an optimal playing opponent
- Time complexity is $O(b^m)$

- Space complexity is $O(bm)$ since dfs.

Minimax is very slow since both b and m are high for reasonable games. This makes the exact solution completely infeasible.

6.2 Resource limit

The standard approach to limit our search, so that it completes given a time and space constraint

- Cutoff test, add a depth limit for the search, maybe even add a quiescence search (searching a big deeper if position is imbalanced)
- Evaluation function that estimates the desirability of position that we apply on the cutoff test

A typical evaluation function is a linear weighted w_i sum of features $f_i(n)$, where

$$E(n) = \sum_i w_i f_i(n)$$

In chess, f_0 can be the difference in number of queens and $w_0 = 9$. Importantly, we should make it so that the evaluation flips if the turn flips.

For minimax, the exact evaluation values don't matter. The behavior is preserved under any monotonic transformations of the eval function. Because only the node value orderings matter, we say that payoffs in deterministic games acts like an ordinal utility function.

To use the cutoff test and evaluate in MinimaxCutoff, we replace the terminal test with cutoff and the utility value by eval. This is however, still quite bad, since the depth/ply lookahead is low.

6.3 Alpha beta pruning

Idea, prune branches of the search tree without influencing the search result.

Define α as the best value that the max player can get so far considering everything else on the path. Let β to be the lowest value (also the best value for the min player) that the min player can get so far considering everything else on the path.

Alternatively, consider the range $[\alpha, \beta]$ at a node. This range implies that currently, the max player can guarantee a score of α while the min player can guarantee a score of β.

Importantly

- At a max level, we will set alpha if the child is higher than alpha. If alpha exceeds beta, we prune the node. This is because min will never play this move since it can already get beta
- At a min level, we will set beta if the child is lower than beta. If beta is lower than alpha, we prune the node. This is because max will never play this move since it can already get alpha

Properties

- Pruning does not affect final result
- Good move ordering improves pruning effectiveness

- With perfect move ordering, time complexity is $O(b^{m/2})$ that essentially doubles the search depth.
- An example for reasoning about which computations are relevant (metareasoning)

Sample code

```
function max(state, a, b)
if cutoff(state)
return eval(state)
end

for child in explore(state)
a = max(a, min(child, a, b))
if a >= b
return b
end
end
return a
end

function min(state, a, b)
if cutoff(state)
return eval(state)
end

for child in explore(state)
b = min(b, min(child, a, b))
if b <= a
return a
end
end
return b
end
```

note that once again, the eval is against the max player.

6.4 Non deterministic games

We can represent chance by creating a chance node to all possible child nodes with the edges weighted by the probabilities. This creates there distinct type of levels: MAX, MIN, CHANCE.

ExpectMiniMax is an algorithm to perfect play non-deterministic games. Importantly, it conducts normal minimax, except that in chance nodes it returns the mean value of its child nodes.

An alpha-beta version also works, but only if the leaf values are bounded and not infinite. This is because averaging with infinities will cause infinities at nodes above that can cause alpha-beta to prune important nodes.

In practice, chance increases the branching factor a lot. Higher depth also reduces the prob of reaching the nodes, meaning that the values of lookahead is diminished. Alpha-beta pruning in such cases with low depth is much less effective.

In expect-minimax, the exact values do matter

- Behavior is only preserved by positive linear transformations of the evaluation function. This is because expected values can change node orderings if the child node values changes non-linearly.
- Evaluation should be proportional to the payoff

7 Machine Learning

Design decisions often need to be fine-tuned for game playing agents. The goal is to automate this tuning.

7.1 Supervised Learning and TD Learning

Some challenges

- Handling each stage of the game separately, opening and endgame books
- Adjust search parameters like depth or breadth of search or different strategies (using search control tuning)
- Adjust weights in evaluation functions
- Find good features for evaluation functions

Book learning has an aim to learn a sequence of moves at important positions

- Use opening book and remember the best move
- Learn from mistakes and check for better alternatives
- Main issue is how to identify important positions (which move out of 100 moves is important)

Search control learning, aim to make search more efficient

- Improve move ordering by generating the moves in a preferred order to maximize pruning of subtrees
- Vary the cutoff depth depend on the complexity of the state

Learning evaluation function weights. Aim to adjust the weights based on real world data to predict the true final utility. Supervised learning consists of a training set, and in the context of weight learning, the attributes are features of a state, and the label is the true minimax utility value. The aim is to find a set of weights so that the evaluation (linear weighted sum of features) more closely approximates the true minimax utility on the training set, and hopefully in practice as well.

Problems with weight machine learning

- Delayed reinforcement. The reward resulting from an action is not realized until several timesteps later, no immediate feedback on the action
- Delayed reinforcement causes the credit assignment problem. We need to know which action was responsible for the outcome since the reward comes afterwards.

Temporal difference learning is for multistep prediction

- Multistep prediction uses previous predictions to predict future values, achieving multiple time series predictions. This is in contrast to supervised learning which is only single step.
- In such problems, correctness of predictions are not known til several steps later. But intermediate steps provide some information about correctness of predictions
- TD learning is a form of reinforcement learning, where agents perform actions in hopes to maximize total rewards that may arrive later

NOT EXAMINABLE. TdLeaf lambda algorithm, temporal difference learning combined with minimax search. The idea was to update the weights in the evaluation function to match the true utility while reducing differences in rewards predicted at different levels of the search true (good eval function should be stable from one move to the next).

The details of supervised learning and TD learning are not examinable. An alternative reinforcement learning algorithm called MCTS will be examined instead.

7.2 MCTS

In some games it is hard to create a good evaluation function

- the material value of pieces are not good predictors of the state outcome as the structure matters more.
- The piece positions can also change considerably throughout the game so that positions are very volatile.
- Branching factor is too high that even alpha beta pruning can only have a small lookahead

In such cases, MCTS is an alternative game search approach that does not depend on a heuristic function and does not require high search depths.

MCTS algorithm

- We iterate the following for steps for each round
- Selection, starting from tree root, use a selection strategy to choose explored child nodes until we reach a node that is not fully explored
- Expansion, expand the node by adding a new child from that node
- Simulation, perform a playout from the newly added node to terminal (ignore moves in the payouts)
- Back-propagation, use the playout outcome, update the statistics in each node in the selection from the root to the newly added node.
- Each node maintains the statistics: number of playouts n and the number of wins w.
- After each round, repeat if we have time remaining
- After no time, choose the child node from root with the most playouts, since our selection strategy should favor exploiting such moves. Picking the highest win rate often selects a lower confidence move with fewer samples

MCTS playouts

- Abandons the use of search with cut-off depth and instead use random playouts

- Instead estimates the average utility of a node by playing multiple games from it all the way to a terminal state using some move selection policy. The value of the state is the average of the outcomes of playouts passing that node.
- Each playout to a terminal state is a simulation
- Typically the average utility of a node over many playouts is the win percentage

Playout policy

- Random moves, pretty bad in practice
- Game specific heuristics from expert knowledge
- Learned evaluation function/policy based on self-play, where it guides moves during playout, but the actual outcome of the playout is still based on the terminal state (not the evaluation)
- With enough simulations, we can measure the average utility of a node from actual game play instead of guessing it using an evaluation function

Comments

- Playout complexity is linear, since we don't search
- Can perform many playouts in the time for alpha-beta to only look a few moves ahead
- Playouts allows us to learn directly from the game rather than some heuristic function
- Focuses more on moves with high win rates but still allowing flexibility in exploring other moves, allows us to make a more informed choice on which move to make
- Likely need to find a good playout policy that picks the best move to play in playouts, or use selfplay datasets

Selection algorithm

- At a node where we've expanded/explored all child nodes, we wish to balance between exploring a good move (exploitation) and exploring an unknown move that hasn't been explored much (exploration).
- A selection strategy from reinforcement learning theory is called the upper confidence bound applied to trees (UCB). For a node, it is

$$UCB = \frac{w}{n} + c\sqrt{\frac{\log N}{n}}$$

  where w is the number of wins, n is the number of explorations, N is the number of parent explorations, and c (often $\sqrt{2}$) is a constant that balances the exploitation (first term) and the exploration (second term).
- The selection strategy using UCB is: select the child node with the highest UCB value.

8 Constraint Satisfaction Problems

For standard search problems, each state is a blackbox in that it can be represented by any data structure with any format. In CSP, we restrict the state representation

- State is defined by a set of variables V with values from a domain D
- Goal test is a set of constraints that specifies the allowable combination of (subset) variable values
- The search aims to find a set of variable assignments to satisfy all constraints
- By restricting the state representation, it allows more general purpose algorithms that performs better than standard search algorithms (like dfs/bfs)

An vertex coloring example

- Find a set of color assignments to node such that adjacent nodes are different colors
- Variables are the nodes, domain contains the set of colors
- Constraints will link every two adjacent nodes with the condition that their colorings are different

Binary CSP

- Binary CSP has each constraint to relate only two nodes (coloring CSP)
- Constraint graphs, only relevant for binary CSP, create an edge between two nodes iff there is a constraint between the nodes. Can be used to identify independent subproblems by solving connected subgraphs

We can also create a constraint graph by creating new constraint nodes that has an edge to all the nodes used in that constraint.

Variables, assuming n variables

- Discrete variables have finite domains, if d assignments, a total of $O(d^n)$ complete assignments.
- Examples are: binary CSP (boolean sat) is NP-complete; job scheduling with variables as the start and end of days is also NP-complete; linear constraints can be solved, but non-linear are harder to solve.
- Typically requires a constraint language that outlines all the constraints
- Continuous variables have infinite domains, like a real start and end time for scheduling.
- Linear constraints can be solved using linear programming method in polynomial time.

Types of constraints

- Unary constraints involve only one variable
- Binary constraints involve only two variables
- High-order constraints involve 3 or more variables
- Preferences (soft-constraints) associates the constraint with a value or cost. It does not have to be satisified (contrary to hard constraints which must be sat), but the goal is to satifisfy the soft constraints with the highest total value or lowest cost. These are constrained optimization problems

Real world CSP

- Assignment problems

- Timetabling problems
- Hardware configution
- Transportation scheduling
- Factory scheduling
- Note that more real world problems have continuous variables

8.1 Standard Search

A naive standard search has the formulation

- Initiate state, empty assigment
- Successor function, assign a value to an unassigned variable that does not conflict with the existing assignments. We backtrack if there are no legal assignments
- Goal test, if the current assignment is complete

Standard search properties

- Works for all CSP
- All goal states are at depth n, so we can use dfs
- Path is irrelevant, so it is a complete-state formulation and we can use complete-state algorithms (hill climb)
- Branching factor $b = (n - l)d$ at level l because there are $n - l$ variables not yet assigned each with d values, hence there are $O(n!d^n)$ leaf nodes. The proof is

$$nd \times (n-1)d \times (n-2)d \times \cdots = n!d^n$$

Backtracking

- Variable assignments are commutative: the order of variable assignments does not matter
- At each level we only need to consider assignments to a single fixed variable, so $b = d$, with $O(d^n)$ leaf nodse
- When we do DFS with CSPs using this single variable assignment, it is call backtracking search
- A basic uninformed search algorithm for CSP, uninformed since it doesn't use the constraint information to dicate searching

You know backtracking, don't need the pseudocode here.

8.2 General purpose algorithm

To improve backtracking, we can consider general purpose methods/heuristics

- Which variables to assign next
- What order to try the values
- Detect inevitable failures early

- Take advantage of the problem structure (shape of the constraint graph)

Minimum remaining values MRV (most constraint variable) heuristic says that we should assign the variable with the fewest legal values. The idea is that we want our search to end early for a failure path.

Degree heuristic is used as a tie breaker for MRV. It states that we should choose the variable with the most constraints on the remaining variables. The idea is that trying those variables first can force a failure state early.

Least constraining value is a heuristic used to determine the order of values to try. It states that we should choose the least constraining value — try the value that rules out the fewest values in the remaining variables first. The idea is that we should avoid finding deadend paths by maximizing options.

Forward checking aims to apply constraints early after every assignment, as to filter out failure paths early. An easy implementation is

- Keep track of the remaining legal values for unassigned variables
- Update the legal values for unassigned variables after every assignment
- Terminates the search when any variable has no legal values

Note that forward checking still doesn't filter out failure paths given the constraints early enough, because it only propagates information from assigned to unassigned variables. Constraint propagation is to repeat the information propagation from assignment that allows us to detect failure states early. A simple method of constraint propagation is arc consistency

- the constraint $X \to Y$ is arc consistent iff for every value of X, there is at least one legal value in Y.
- the idea of constraint propagation using arc consistency is to ensure that all remaining constraints are arc consistent, by eliminating impossible unassigned variable values
- If the arc consistent problem has any unassigned variable with no legal states, fail early

Arc consistency algorithm

- Loop over all constraints
- For each constraint $X \to Y$, remove all values in X can has no legal value in Y.
- After removing a value to X, we must recheck constraints directed to X
- Can be ran as a preprocessing step or during the backtrack

AC3 algorithm, takes a csp and returns another csp with reduced domains

```
function AC3(csp)
 queue = arcs in csp
 while queue is not empty  // O(n^2)
  (x, y) = pop(queue)
  any = remove_inconsistent_values(x,y)
  if any
   for z in neighbours(x)  // O(d)
    queue.add(z,x)
```

```
 return new csp with reduced domain

function remove_inconsitent_values(x,y)
 removed = false
 for x in domain(x) // O(d)
  if no value y in domain(y) allows (x,y) to sat // O(d)
   remove x from domain(x)
   removed = true
 return removed
```

Time complexity of AC3

- $O(n^2d^3)$ where n is the number of variables and d is the domain size. A rough proof is in the pseudocode
- Newer algorithms can reduce it to $O(n^2d^2)$

Independent subproblems can be identified by connected components. These independent subproblems can be solved separately

- If each subproblem has c variables out of n total, the time complexity is $O(\frac{n}{c}d^c)$ which is linear in n if c is small
- However, completely independent subproblems are rare. But there are specific graph structures for problems (like trees) that forms independent subproblems

Tree-structured CSP has a constraint graph that is a tree with no cycles. The theorem is that tree csp can be solved in $O(nd^2)$ linear time. The algorithm is

- Choose a variable as root, order variables into a list such that each node's parent precedes it in the list
- From j from n to 2 (last to root), make $parent(x_j) \to x_j$ arc consistent
- From j from 1 to n, assign x_j randomly given that it is consistent against parent node
- The algorithm takes $O(nd^2)$ time because each arc consistency takes d^2 time and we repeat that n times

Nearly tree structured csp, use conditioning

- Conditioning means to assign values to a subset of variables, then to prune them from the graph, so that the resulting variable constraints forms a tree
- Cutset is a set of variables that can be deleted so the constraint graph forms a tree
- Cutset conditioning is to try all arangements in the cutset for conditioning, then trying to solve the remaining tree csp
- Cutset conditioning with a FOUND cutset size c takes $O(d^c(n-c)d^2)$, which is good if c is small

Iterative algorithms or local search

- Local search is an algorithm for complete-state problems, where it assigns all variables, and change one variable assignment at a time

- For CSP, create an initial state with unsat constraints, and repeatedly reassign variable values
- Variable selection is the variable to consider, value selection is the value to assign to the variable.
- Variable selection step should randomly select a conflicted variable
- Value selection should be done in respect to a min-conflict heuristic (ie, total number of violated conflicts), where we choose the value that minimizes this min-conflict heuristic

Iterative min-conflict performance

- Given a random initial state, can solve the CSP in almost constant time for arbitrary n with high probability.
- Applies for any randomly generated CSP problem/constraints except those within a narrow range of the ratio

$$R = \frac{\#\text{constraints}}{\#\text{variables}}$$

9 Uncertainty and Probability

Some problems cannot be solved by pure logic. This may be due to

- Partial observability of the state space
- Noisy sensors and observations
- Uncertainty in action outcomes
- Modelling complexity

In such cases, a logicial solution either

- Risks falsehood (incorrectness) by expressing certainty
- Leads to conclusions that are too specific and weak (conditioning on a lot of variables)

Solutions

- Nonmonotonic logic, assumes a list of things to make a conclusion, and change conclusion if evidence breaks assumptions. Problems are that: which assumptions are reasonable and which evidences are needed to break assumptions
- Rules with confidence factors. For each observation result pair (rule), assign a confidence factor (number) on whether the observation correlates with the result. The problems are that the direction of causalty is omitted when combining multiple rules.
- Probability, given the available evidence, assign a probability to the event

Note that Fuzzy logic is NOT probability. It handles the degree of truth and not the uncertainty in the truth.

9.1 Basic Probability

Probability

- Summarizes the effects of: laziness in enumerating all scenarios, and ignorance in the initial conditions and relevant state spaces
- Two kinds of probabilities: Subjective and Bayesian probability
- Bayesian probability relates the likelihood of a proposition given one's own state of knowledge. Bayesian probabilities does not equal the exact probability of the event always, rather, it is an estimate that is updated on new pieces of evidence

Decisions under uncertainty

- The decision made depends on one's preference between two trade offs
- Probability theory handles likelihood of events
- Utility theory is used to handle preferences
- Decision theory combines utility theory and probability theory to model decision making

Probability revision

- The sample space Ω has the set of outcomes
- A probability space is a sample space with a probability function from subsets of the sample space to real numbers $P : \mathcal{P}(\Omega) \to \mathbb{R}$
- A subset of the sample space is an event $A \subseteq \Omega$.
- Axioms

$$\begin{aligned} P(A) &\geq 0 \\ P(A \cup B) &= P(A) + P(B) \\ P(\Omega) &= 1 \end{aligned}$$

- Random variables is a function from outcomes to numbers, $X : \Omega \to \mathbb{R}$. A probability distribution $f(x)$ for a random variable is the probability that X is a particular value

$$f(x) = P(X = x) = \sum_{\omega : X(\omega) = x} P(\omega)$$

9.2 Proposition under probability

Proposition

- A proposition in probability is the event where the proposition is true
- Can model propositions events a, b as boolean random variables A and B. To translate between logicial operators to probability operators

$$\begin{aligned} a &= \omega : A(\omega) = 1 \\ \neg a &= \omega : A(\omega) = 0 \\ a \wedge b &= \omega : A(\omega) = B(\omega) = 1 \end{aligned}$$

- Often, the sample space is defined by the values of a set of random variables (not the other way around)

- A proposition can be destructed into a disjunction (OR) of atomic events where the prosposition is true. This is equivalent to the disjunctive union of their boolean random variables in probability.
- All theorems in proposition/logic translates to probability theorems, vice versa.

Consequences of proposition in probability

- Since $P(a \vee b) = P(a) + P(b) - P(a \wedge b)$, logicially related events/propositions must have probabilities that are related (correlated)
- A theorem that if a betting agent doesn't follow these probability axioms, we can force it to make bets where they lose no matter the random outcome

Proposition syntax and terminology use to model queries in the environment

- Propositions are boolean random variables
- Discrete random variables (can be finite or countably infinite), with domain values being exhaustive and mutually exclusive. The domain values are in angled brackets
- Continuous random variables (bounded or unbounded) have uncountably infinite domain
- An arbitrary combination of boolean propositions is also a proposition

Probability terms

- Prior or unconditional probability is the belief prior to new evidence
- Marginal probability distribution for a rv is the set of probabilities all the rv's atomic values, ignoring the other rvs
- Joint probability distribution for a set of rvs gives the probabilities for each combination of atomic rv values
- We use a regular P for a probability function, and a bolded $\mathbf{P}$ for a probability distribution
- Every question about a domain (which is the combination of rv values) can be answered by the joint distribution

To compute the marginal probabilities, we sum over the probabilities on all combinations of the unimportant rvs. For continuous random variables, the probabilities are densities: uniform or gaussian.

Conditional probabilities

- Posterior probability is the probability of a proposition given a set of evidence, $P(A|B, C)$. This is our beliefs if we only know the prior probability of the proposition and the new evidence
- Note that less informative beliefs remains valid even if new evidence arrives (since they are not conditioned on the new evidence), but they may not be useful
- New evidences can be irrelevant which don't change the posterior, so we can remove them in the conditional. This requires domain knowledge and can massively simplify inference (computing the posterior probability).

Conditional probabilty rules

$$\begin{aligned} P(A|B) &= \frac{P(A \cap B)}{P(B)} \\ P(A \cap B) &= P(A|B)P(B) = P(B|A)P(A) \\ P(X_1, X_2, \ldots, X_n) &= P(X_1)P(X_2, \ldots, X_n|X_1) \\ &= P(X_1)P(X_2|X_1)P(X_3|X_1, X_2)\ldots P(X_n|X_1, X_2, \ldots, X_{n-1}) \end{aligned}$$

9.3 Inference By Enumeration

Inference by enumeration

- A technique that computes the probability of a proposition by summing over the the events where it is true
- For proposition ϕ, we sum over all atomic outcomes ω where ϕ is true

$$P(\phi) = \sum_{\omega} P(\omega)$$

- Lmao dude this is just basic probability

To compute the posterior probability, we use

$$P(X|Y) = \frac{P(X, Y)}{P(Y)}$$

Typically we can treat the denominator as a normalization constant α, so that

$$P(X|Y) = \alpha P(X, Y)$$

This nets us a conditional probability distribution on the query variable X, if we fix the evidence variables Y and sum over the hidden variables (not shown) against the query variable value.

Define all the variables as X, the query variables as Y, evidence variables E, and the hidden variables as $H = X - Y - E$. Our goal is to find the posterior joint distribution of Y given the evidence variables. The formula is

$$P(Y|E = e) = \alpha P(Y, E = e) = \alpha \sum_{h} P(Y, E = e, H = h)$$

The problems are that

- The worse case time complexity is $O(d^n)$ where d is the max values for a hidden variable, and n is the num of hidden variables
- The space complexity is $O(d^n)$ to store the joint distribution including the hidden variables
- How to compute the hidden joint distribution?

9.4 Conditional indepedence

Define A and B to be independent iff $P(A, B) = P(A)P(B)$. For distributions where most variables are independent, this reduces the entries needed for a joint distribution table to linear. However, absolute independence is rare in practice.

Define A to be conditionally independent to B given C iff

$$\begin{aligned} P(A|B, C) &= P(A|C) \\ P(B|A, C) &= P(B|C) \\ P(B, A|C) &= P(B|C)P(A|C) \end{aligned}$$

Notice that A cond indep to B given C is equivalent to B cond indep to A given C.

Practically, conditional independence reduces the size of the joint distribution to linear in n the number of hidden variables. Conditional independence is a powerful and robust method to use domain knowledge to simplify uncertain environments.

Bayes' rule

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} = \alpha P(B|A)P(A)$$

Used to compute diagonistic probability (likelihood to have a disease given symptom) from causal probability (probability of symptom given disease), where A is the disease and B is the symptom.

A naive bayes model uses bayes rule and conditional independence to model the joint distribution

$$P(Y, X_1, X_2, X_3) = P(X_1, X_2, X_3|Y)P(Y) = P(Y)\prod_i P(X_i|Y)$$

assuming that the effects X_i are conditionally independent with each other given the cause Y. This reduces the joint probability table to linear against the number of variables.

9.5 Example

Consider a grid of cells, the rv for a blackbox in a cell is B_{ij} and the rv for a pulse in the cell is P_{ij}. To compute the probability distribution of the blackboxes given the pulses, we need to solve

$$P(B_{ij} \dots | P_{ij} \dots)$$

This is equivalent to

$$\begin{aligned} P(B_{ij} \dots | P_{ij} \dots) &= \alpha P(P_{ij} \dots | B_{ij} \dots)P(B_{ij} \dots) \\ &= \alpha' P(P_{ij} \dots | B_{ij} \dots) \qquad \alpha' = \alpha P(B_{ij} \dots) \\ &= \alpha' \prod P(P_{ij}|B_{ij} \dots) \qquad \text{since } P_{ij} \text{ are cond indep with each other on } B_{ij} \\ P(P_{ij}|B_{ij} \dots) &= P(P_{ij}|\text{known searched}, \text{boundary searched}) \end{aligned}$$

With the last step removing B_{ij} that are far away from the pulse P_{ij}, as the pulse of a cell is cond indep on blackboxes in far away locations given that we know if the blackboxes are in nearby locations.

This allows us to massively simplify the query to approximately $O(nm)$ where n is the grid cells and m is known plus boundary blackbox squares.

10 Bayesian Networks

Bayesian networks

- A graphical notation for conditional independence assertions that leads to a compact specification of the full joint distribution
- Nodes for variables
- Directed acyclic graph, where an edge implies influence from the parent to the child
- A conditional distribution for each node given all its parents

  $$P(X_i|\text{Parents}(X_i))$$

  Sometimes this is represented as a CPT (conditional probability table), outlining all conditional probabilities

Network topology

- A node is conditionally independent to any non-child nodes given its parents
- A node is independent to nodes that is not its ancester or child.
- A parent has an edge to a child if it has a causal relationship with the child. It can also have an edge if it is related to the child such that conditionating on other variables still don't make them independent.

In most of the cases, we build the bayesian network based on casuality assumptions.

Global semantics is the rule to compute the joint distribution given the bayesian network

$$P(X_1, X_2, \ldots, X_n) = \prod_i P(X_i|\,\text{parents}(X_i))$$

Local semantic dicates that every node is conditionally independent to non-child nodes given its parents

$$P(X_i|X_j, \text{parents}(X_i)) = P(X_i|\,\text{parents}(X_i))$$

There is a theorem that states: local semantics in a bayesian network implies global semantics, and vice versa.

Compactness of bayesian networks

- Assuming boolean variables, a CPT of a variable X with K parents has $O(2^k)$ rows for the combination of parent values. Each row requires only one value for when X is true
- This means that the whole network requires $O(n2^k)$ numbers if k is the maximum parent size, which is linear against n compared to the $O(2^n)$ for the full joint distribution.

Given a series of locally testable conditional independence assumptions, we can build a bayesian network that guarantees the global semantics. The process is

- Choose an ordering of variables $X_1, \ldots, X_n$
- For each i from 1 to n, add X_i to the network, and select the parents from $X_1, \ldots, X_{n-1}$ such that

  $$P(X_i|\,\text{parents}(X_i)) = P(X_i|X_1, \ldots, X_{n-1})$$

- This will guarantee global semantics since

$$P(X_1, \ldots, X_n) = \prod_i P(X_i | X_1, \ldots, X_{n-1}) = \prod_i P(X_i | \text{parents}(X_i))$$

Ordering in bayesian network construction

- Deciding conditional independence is hard if non-causal directions
- Causal models are easier for humans to build
- We should order the variables in building the network in causal order. Otherwise, the network is less compact

To use a bayesian network

- Construct a network
- The initial evidence is a variable
- Some variables are testable, some are problem variables
- Hidden variables can be used to reduce the degrees of nodes

Inference tasks using a bayesian network

- Simple queries computes the posterior marginal, $P(X|E = e)$
- Conjunctive queries is the joint posterior marginal, $P(X_i, X_j|E = e) = P(X_i|E = e)P(X_j|X_i, E = e)$.
- Value of information, which evidence to seek next
- Sensitivity analysis, which probability values are the most important in inference
- Explaination, can trace the network output to get the reasoning
- We only focus on simple queries and conjunctive queries

Inference by enumeration

- An intelligent way to sum probabilities from the joint distribution without constructing the table
- Can handle simple and conjunctive queries
- First use the conditional probability rule to convert the conditional prob to two joint distribution.
- Then ignore the denominator, for the numerator, sum over the hidden variables
- Convert the joint distribution inside the sum into a product of the CPT entries, using global semantics.
- We can simplify the sum over the hidden variables by moving constant terms out of the sum
- Can use recursive DFS, requires $O(n)$ space and $O(d^n)$ time.

- Example

$$\begin{aligned} P(A|B,C) &= P(A,B,C)/P(B,C) \\ &= \alpha P(A,B,C) \\ &= \alpha \sum_d P(A,B,C,D) \\ &= \alpha \sum_d P(D)P(B|A,D)P(C|A,D) \end{aligned}$$

Variable elimination

- Some products in enumeratioin are computed multiple times across multiple sums. This can be optimized by caching these factors.
- In general, variable elimination will do the summation right-to-left, storing the intermediate factors (a factor ends at a summation sign) to avoid recomputation
- The two operations are: summing out a product of factors (move constants out, and use pointwise multiplication to sum out the remaining factors)

$$\sum_x f_1 f_2 f_x = f_1 f_2 f_{\bar{x}}$$

, and a pointwise product

$$f_1(x,y)f_2(y,z) = f_3(x,y,z)$$

- Irrelevant variables are variables that does not contribute to the query, and we can remove them from the formula. Theorem is that Y is irrelevant unless if Y is an ancestor to the known variables.

11 Decision Theory

Key insights on decision making

- Our intuitions often mislead us under uncertainty
- We require formal tools to reason about uncertain decisions

Decision Theory

- Logical agents only concerns about binary goals (achieved or not), while decision-theoretic agents assign utility values to a range of outcomes to handle trade-offs and uncertainties.
- Probability encodes what the agent currently believes, utility encodes what the agent currently prefers. Decision is made by combining beliefs (probability) and desires (utilities)

11.1 Utilties and Preferences

Utilities

- A function that maps outcomes to real numbers that describes the agent's preference
- Denoted as $U(s)$ where higher utility means more preference. Therefore it numerically captures the agent's goals/preferences

- Derived from: explicit designs by agent designers based on task domain knowledge, learnt from experience and feedback, or derived from preferences where they are consistent with the agent's ranking of outcomes

Prizes and lotteries

- Prizes are fixed outcomes denoted by a symbol A
- Lotteries consists of multiple uncertain outcomes denoted in probability by $[p, A; 1-p, B]$.

Expected utility is the utility of a lottery

$$U([p_1, S_1; \ldots; p_n, S_n]) = \sum_i p_i U(S_i)$$

Rational preferences

- A rational agent must be able to compare any two outcomes (completeness)
- It must be consistent in its preferences (transitivity)
- Irrational agents can make contradictory or exploitable decisions
- Denote $A \to B$ as A having a higher or equal utility than B. Denote $A = B$ if it is indifferent.

Rational preference axioms/constraints

- Orderability, $A \to B \vee B \to A \vee A = B$ is true
- Transitivity, $A \to B \wedge B \to C \implies A \to C$
- Continuity, $A \to B \to C$ implies that there exists at least a p where $[p, A; 1-p, C] = B$.
- Substituability, $A = B \implies [p, A; 1-p, C] = [p, B; 1-p, C]$
- Monotonicity, $A \to B$ implies that if $p \geq q$, $[p, A; 1-p, B] \implies [q, A; 1-q, B]$
- Decomposability, $[p, A; 1-p, [q, B; 1-q, C]] = [p, A; (1-p)q, B; (1-p)(1-q), C]$

Theorem of utility: given rational preferences that satisfies the rational preference constraints , there exists a real-value utility function U such that

$$U(A) \geq U(B) \iff A \to B \vee A = B$$
$$U([p_1, S_1, \ldots; p_n, S_n]) = \sum_i p_i U(S_i)$$

Essentially, a rational set of preference will always have a valid utility function.

Violating rationality in preferences

- Leads to self-evident irrationality
- An agent with intransitive preferences can be induced to give away all its money by continuously moving it to more preferrable choices while taking a fee in the middle

Decision with expected utility

- Choose action

- Compute expected utility on the outcomes that the action leads to each with a utility and probability
- Choose the action with the highest EU
- Namely
$$a^* = \arg\max_a \sum_{s \in S} P(s|a)U(s) = \arg\max_a EU(a)$$
- This is the MEU principle, the maximum expected utility principle (where we choose the action with the maximum expected utility)

Constructing utility functions

- Not always obvious to define a utility function
- Direct elicitation, ask the users to rank outcomes
- Trade-off analysis, present choices that forces a preference between competing outcomes
- Observation, infer preferences from behaviors

Standard approach of accessing human utilities

- Used to rank outcomes and get utility values
- Compare a given state A to a standard lottery L_p that offers the best possible prize (1000 dollars) with prob p, or the worse possible catastrophe (death) with probability $1 - p$.
- Adjust p such that $A = L_p$
- Rank utility of states using the standard lottery probabilities

Utility scales

- Normalized utilities, best utility is 1, worst is 0
- Micromorts, one-millionth chance of death. Used to evaluate risks in hazardous scenarios
- QALY, quality adjusted life years, how many years (adjusted by quality) does the patient have left. Used for medical decisions involving substantial risks
- If outcomes are all deterministic prizes, only ordinal utiltiies can be determined (scale doesn't make sense so only the ordering matters)
- Agent behavior under MEU is invariant with positive linear transformations of utility functions
$$U'(s) = mU(s) + c, m > 0$$

Utility of money

- Typically not linear
- Diminishing returns, each additional dollars adds less utility
- St Petersburg paradox presents a lottery with infinite expected monetary value but with low perceived utility

Normative vs descriptive decision theory

- Normative theory concerns about rational agent behavior/decisions. It assumes consistent, utility-maximizing behavior
- Descriptive theory describes how real people makes decisions. Influenced by biases, emotions, and framing, where unrelated factors can change the agent's behaviors and make it irrational

11.2 Decision Network

Decision network (influence diagram) are extensions of belief networks. It consists of

- Chance nodes (ovals) that represents our uncertainty, similar to bayesian belief networks
- Decision nodes (rectangles) represents possible choices
- Utility nodes (diamonds) represents utilities/preferences that we can deduce the optimal action from
- Utility node is often connected to both decision nodes and chance nodes. It maintains a table that maps each decision and chance state to a utility.

Forward algorithm to compute optimal action

- For each set of action of all decision nodes, compute the total expected utility of all utility nodes based on the action and chance node cpt
- Formula for expected utility is identical to the EU formula but fixing the action value
- Select the action that maximizes total expected utility

Rollback method, outcome trees

- Another algorithm for evaluating decision networks
- Unroll the network into a decision/outcome tree, where each level conditions a value on a chance node (circle) or a decision node (triangle)
- The utilities are leafs (squares) at the end given the path taking with conditioned values
- Equivalent to expectedminimax where the leaves are utilities
- Solve by starting from the bottom: for each chance node compute the expected utility given the child expected utilty, for each decision node take the maximum child expected utility. The path from root that leads to the highest expected utility contains the set of optimal actions

11.3 Value of information

VOI

- Decision making often involves knowing which questions to ask
- Gathering evidence reduces uncertainty but has a cost
- Information value theory enables agents to compute which information to acquire based on their value
- Without information, use the prior. With information, use the posterior

Value of perfect information

- Given that we have a forecast of some nodes, the value of perfect information is the difference in maximum EU given the forecast and maximum EU without the forecast given the optimal actions.
- Note that we need to include the forecast chance node in the maximum EU without forecast calculation
- Helps to assess whether paying for the forecast is worthwhile. Also helps to rank forecasts
- VPI is always non-negative since we can always increase the expected utility given new information

Intuition in value of information

- Information is only valuable if it is likely to chance the optimal decision and that the stakes are high (increase in EU)
- Consider two actions, each yielding the expected utility of U_1 and U_2. There exists a random evidence E that we can observe. Let the random expected utility of these actions conditioned on the evidence have the probability distribution $P(U_1 = x|E)$ and $P(U_2 = x|E)$.
- There are only three scenarios in the value of information given the conditional distribution of the two actions
- High confidence, high stakes, no value, since we are very confident that the orderings will not change
- High uncertainty, high stakes, valuable, since the ordering can change and the loss in potential expected utility is very high.
- High uncertainty, low stakes, little value, since the ordering can change but the loss in potential expected utility is low
- High confidence implies that we are confident that the ordering between the EU of the two actions is changed given any evidence value. The prob distributions are far and don't overlap
- High stakes implies that given the worst possible evidence, the loss in potential expected utility is high. The two probabilities overlap and the distributions are wide so that it is possible for the currently higher U to end up low while the currently lower U to end up high.

12 Robotics

Key terms: percepts, actions, effectors, sensors

- Percepts are observations made by sensors
- Actions are moves made by effectors

Manipulators

- Manipulators are effectors, components that performs actions

- The dof is a set of dimensions that defines the possible configurations of the robot (the underlying state space). The robot at any point is at a point of the c-space
- Can use effectors to move in c-space
- We need at least 6 DoF to position end-effectors arbitrarily. For systems that are dynamic, add 6 DoF more (one for each original degree) for velocity

Non-holonomic

- Robots with less controls than degrees of freedom. Dimensions of controls less than dimensions of DOF.
- Cannot generally transition between two infinitesimally close configurations

12.1 Uncertainty

Two major sources of uncertainties for any robot

- Uncertainty in percepts
- Uncertainty in actions

Percept uncertainty

- For human vision: small region with color; most fov is monochrome/low-resolution/motion-sensitive; mostly 2d images
- Human vision breaks down in hallucinations and optical illusions. So our sensors does not capture the full configuration

Action uncertainty

- Effectors will not perform an action exactly as required with infinite precision.
- Reasons for uncertainty: slippage, inaccurate encoder, rough surfaces, obstacles, effector breakdown
- Action uncertainty will accumulate without bound. Even starting with perfect knowledge and moving using actions with very small error, after a set of infinitely long action sequence the system will have infinite error in its configuration space estimate (or the c-space error approaches infinity as action sequence increases)
- Can be dealt with by using sensors to verify actions

Sensors

- Range finders to find distances
- Imaging sensors like camera
- Proprioceptive sensors that verify actions, decoders

Implication of sensor uncertainty

- Need to make assumptions about how the world/robot behaves to interpret sensor readings, since sensors are not perfect
- Assumes that a finite resolution sampling is sufficient to detect the obstacles. (Need higher resolution if obstacles are spiky and with sparsely distributed pins)

- Assume the structure of the robot is unchanged to interpret the readings (like where the obstacle is and where the sensor on the robot is)
- Assumes that the sensor readings only has a limit probability of being correct.

Example distance sensor uncertainties

- Number corresponding to the nearest obstacle in straight line path, up to some distance upperbound
- Known precision error in distance
- Known false positive or false negatives on whether an obstacle exists
- Has small spatial extent (cannot work if partially obstructed)
- Finite time so cannot check all directions at once. Only a set of readings close in time can communicate information

Algorithm to solve both uncertainties

- Localization, given map and landmarks, update pose distribution
- Mapping, given pose and landmarks, update map
- Simultaneous localization and mapping (SLAM), given observed landmarks, update pose and map at the same time
- Essentially SLAM alternates between localization and mapping. It does localization if it senses a change in pose. It does mapping otherwise assuming that the pose changes. Uses probabilistic formulations

Localization

- Given a set of actions, states, and observations
- Localization computes the current location and orientation (pose/state) given the observations
- Update X_{t+1} given action A_t and observation X_t and past state X_t
- Sensor model uses observations of landmarks to estimate the exact state of the robot
- Motion model updates the state using its movements in a small period of time
- A localization method uses both the sensor and motion model to predict its location. Will assume gaussian noise in sensor measurements and thus motion predictions.

Particle filtering

- An algorithm for localization without landmarks
- Sample positions from the uniform prior robot position
- Update likelihood of each sampled position using sensor measurements
- Update posterior distribution of robot location using likelihoods of the sampled positions
- Resample positions from the updated posterior distribution of robot location and repeat

Localization algorithm with landmark

- Need to continuous update our distribution for the current state given the latest measurements
- Uncertainty in robot's state increases until it finds a landmark (an object with know position), then it will decrease
- Assumes that landmarks are identifiable in observation. Otherwise, the posterior distribution after seeing a landmark will be multimodal

Mapping

- Robot will see landmarks when performing actions
- There will be uncertainty in the location of the robot after moving, and also uncertainty in the location in the landmarks observed. This grows overtime
- After observing the a previously seen landmark after a path/loop, we can complete the mapping reducing all uncertainties to zero assuming that initial landmark is unmoved.

12.2 Incremental bayesian inference

Simplified problem: determine whether an obstacle is there given continuous measurements from a sensor

- Use bayesian inference: find probability of hyp given evidence
- Bayes theorem for a single measurement is

  $$P(H|M) = \frac{P(M|H)}{P(M)} P(H)$$

  where the $P(H)$ is prior location dist, $P(M|H)$ comes from our sensor model/assumptions to handle sensor uncertainty, $P(M)$ is a normalization factor, and $P(H|M)$ is the posterior probability.
- Incremental bayesian inference extends a single measure update to multiple measurements. Given a set of conditionally independent measurements given H, find the probability of the hypothesis
- This incremental form will iteratively use a new measurement to update the current $P(H)$. This bayesian update rule is

  $$P(H) = \frac{P(M|H)}{P(M)} P(H)$$

  note that the only number that changes over the iteration is $P(H)$ (and thus $P(M)$). The likelihood is constant.

Incremental bayes derivation

$$P(H|M_1, M_2) = \frac{P(M_1, M_2|H)P(H)}{P(M_1, M_2)} \tag{1}$$

$$= \frac{P(M_2|H)P(M_1|H)P(H)}{P(M_1)P(M_2|M_1)} \tag{2}$$

$$= \frac{P(M_2|H)P(H|M_1)}{P(M_2|M_1)} \tag{3}$$

$$= \frac{P(M_2|H)P(H|M_1)}{P(M_2|H, M_1)P(H|M_1) + P(M_2| \wedge H)P(\wedge H|M_1)} \tag{4}$$

and we can expand the denom using law of total prob against H,and removing M_1 in the conditioning due to conditional independence

Localization and mapping uses incremental bayes in a higher dimension version.

12.3 Motion planning

Motion mapping

- Aims to use actions and effectors to move the robot to the desired location
- Plan in the configuration space, restricted by the robot's degree of freedom (only points in c-space that can be reached by dof). A solution is a point trajectory in this free c-space.
- The continuous variant will have infinitely many states as the c-space is continuous. Two methods to convert to finite state space are: cell decomposition, skeletonization

Cell decomposition

- Divide free space into geometrically simple cells, where traversing within cells (and between neighbor cells) are easy
- Solution is the shortest path through these cells.
- There may be no path in the current set of pure cells (every point is reachable inside). Solution will be to recursively decompose mixed cells with finer resolution
- Can generate jaggy motion due to finite resolution. A variant called octomap will vary the resolution when closer to the obstacles

Skeletonization

- Identify finite number of points in free space that are easily connected in c-space that forms a graph where any two points are connected.
- Can randomly sample space and keeping those in the free space, creating a probabilistic roadmap. Although we need to generate enough points to ensure that most pairs are connected in the graph
- Or use a voronoi diagram where the points are all maximum equidistance from obstacles and the path still exists, which creates a safer path away from obstacle points but loses speed. Implementation can use dilution of the configuration space or bruteforce scanning. Often takes a long time to compute and don't scale well into higher dimension.

Both cell decomposition and skeletonization can be done prior to inference since they are often expensive. This requires setting the resolution of the building algorithms.

QA

- Are the agent methods useless due to uncertainty?
- Most problems don't have uncertainties: planning, game playing
- Can incoporate a standard agnet to a system and handle uncertainty externally: replan if assumptions changed
- Can apply uncertainty to the agent: bayesian inference
- Although uncertainties can create some problems that cannot easily be handled by conventional algorithms