# Artificial Intelligence: A Model Approach

## Intelligent Agent

### Terminology

#### Agent

Anything that can perceive it’s environment through sensors and act upon that environment through actuators

#### Percept

The agent’s perceptual inputs at any given instance

#### Percept Sequence

The complete history of everything the agent has perceived

### Concept

#### Properties of task environment

* + - * 1. Fully observable vs. Partial observable

Based on whether an agent’s sensors perceive complete states of the environment relevant to the choice of action

* + - * 1. Single agent vs. Multi-agent
        2. Deterministic vs. Stochastic vs Non-deterministic

Unfamiliar point

Deterministic environment is predictable based on the current environment state and the action executed by the agent.

Stochastic environment is unpredictable

Non-deterministic environment is one in which actions lead to one of several possible outcomes. Since all of these outcomes are predictable, the agent should have plans for each of them in order to maximize the performance measure

* + - * 1. Episodic vs. Sequential

Unfamiliar point

In episodic environment, the current decision totally depend on the current environment state and have no further effect on future decisions

In sequential environment, the current decision affect all future decisions

* + - * 1. Static vs. Dynamic

Unfamiliar point. Based on whether the environment changes while the agent is deliberating

In dynamic environment, the agent needs to keep an eye on the environment while it’s deliberating.

In semi-dynamic environment, the environment itself stays the same, but the performance measure varies while it’s deliberating

* + - * 1. Discrete vs. Continuous

Based on whether the environment states can be divided into different frames

* + - * 1. Known vs. Unknown

#### Classes of agents

* + - * 1. Simple reflex agents

Which simply use condition-action rule

* + - * 1. Model-based reflex agents

Which keep track of the percept history and thereby infer the unobserved aspect of the current state, then apply condition-action rule

* + - * 1. Goal-based agents

There is no condition-action rule anymore, a goal-based agents has a goal and act towards it. It now considers question like "what will happen if I do such-such-such?"

* + - * 1. Utility-based agents

Goals now are not certain states, they are measured by how desirable the environment states are, we call this utility. An agent’s utility function is its self performance measure and a rational agent may need to reckon the tradeoff(desired states may not be able to be achieved simultaneously) and even probability(desired states may not be able to be achieved with certainty) to maximize the utility

* + - * 1. Learning agents

Unfamiliar point

Performance element is what we previously considered to be the entire agent

Critic evaluates the performance measure and send feedback to learning element. Notice that critic receives inputs from sensors rather than performance element so as to avoid self-delusion

Learning element is responsible for making improvements

Problem generator is responsible for suggesting actions that will lead to new and informative experiences, i.e., suggesting to try something unknown to see if it might lead to a better outcome

Apart from perceiving environment and acting upon them(that’s what performance element do), a learning agent evaluates the performance measure in critic, makes improvements in learning element, and suggests to try something new in problem generator

#### Representations of environment states

* + - * 1. Atomic representation

Each state is indivisible, i.e. without internal structure

* + - * 1. Factored representation

Each state is split up into a fixed set of variables or attributes

* + - * 1. Structured representation

Each state includes objects and relationships between them

### Q & A

#### What kind of agent is intelligent?

* + - * 1. An intelligent agent is rational—it maximizes its performance measure based on its experience (percept sequence to date, BTW, this indicates that the agent might need the ability to learn) and inherence (prior knowledge of the environment)

#### What’s the difference between agent function and agent program?

* + - * 1. The agent function is an abstract mathematical description; The agent program is a concrete implementation, running within some physical system

#### How to devise an appropriate performance measure?

Unfamiliar point

* + - * 1. As a general rule, it’s better to design one according to what one actually wants in the environment, rather than according to one thinks the agent should behave. After all, a performance measure is an evaluation of any sequence of environment states

It’s crucial to realize environment states, not agent states used above. FMI, see P37

#### What does the task environment include?

* + - * 1. Performance, Environment, Actuators, Sensors, or for the acronymically minded, PEAS

## Solving Problems by Searching

### Terminology

#### Completeness

An complete algorithm is guaranteed to find a solution when there is one

#### Optimality

An optimal algorithm always finds an optimal solution

#### Effective branching factor b\*

If the total number of nodes generated by A\* for a particular problem is N and the solution depth is d, then b\* is the branching factor that a uniform tree of depth d would have to have in order to contain N+1 nodes. Thus,

N + 1 = 1 + b\* + (b\*)^2 + … + (b\*)^d

In other words, N + 1 nodes compose a complete b\*-ary tree of depth d

### Concept

#### System

Unfamiliar point

* + - * 1. Open-loop system

A system which ignores the feedbacks and therefore breaks the loop between agent and environment

* + - * 1. Close-loop system

A system which keep communicate with its environment and make decisions based on it

#### Heuristic

* + - * 1. Admissibility

An admissible heuristic is one that never overestimates the cost to reach the goal

* + - * 1. Consistency

Consistency (or monotonicity) emphasizes that h(n) ≤ c(n, n’) + h(n’), where c(n, n’) is the cost from n to n’. AKA triangle inequality, which stipulates that each side of a triangle cannot be longer than the sum of the other two

* + - * 1. What’s the relation between admissibility and consistency?

A consistent heuristic is also admissible, but not vice versa. In other words, consistency implies admissibility or admissibility is a superset of consistency

* + - * 1. Given two admissible heuristics, how to tell one is better than the other?

The greater one is better because the heuristic is designed to be approximate, as close as possible, to the actual cost from the current node to the goal. If h1 ≥ h2, then we say h1 dominates h2.

* + - * 1. If we have a collection of admissible heuristics h1, …, hm is available for a problem and none of them dominates any of the others all the time, which should we choose?

We can define h(n) = max{h1(n), …, hm(n)}. Since none of hs overestimate the cost, neither will the maximum.

This solution has one drawback: it computes every heuristic when calling max, which could be very time-consuming.

* + - * 1. How to generate heuristics by agent itself?

Unfamiliar point

Generate heuristics from relaxed problem, which has fewer restrictions on the actions compared to the original problem. This is equivalent to adding additional links to the state space

Generate heuristics from subproblems: pattern database

Pattern database

Unfamiliar point. Reduce the problem to a simpler subproblem, and store exact costs for every possible subproblem instance to form a database. Such costs are constructed by searching back from the goal and used as heuristics later.

How to improve the accuracy of pattern database?

Unfamiliar point

Reduce the problem to several simpler subproblems, and construct a database for each of the subproblem. Each database yields an admissible heuristic, and these heuristics can be combined by taking the maximum value.

If the pattern databases are disjoint, a new heuristic can be generated by adding costs from the disjoint pattern databases

Pattern databases are disjoint if an action only affects one subproblem

Generate heuristics from experience

### Algorithm

#### Uninformed search

* + - * 1. Breadth-first search
        2. Uniform-cost search

A greedy algorithm using g to order the priority queue, where g is the lowest path cost from the initial node to the current

* + - * 1. Deep-first search

The only search listed here which is not optimal and incomplete

* + - * 1. Iterative deepening search

What’s the advantage of IDS (iterative deepening search)?

To save space. It takes only O(bd) memory

where b is breadth, d is the depth of the shallowest solution

It’s complete, even for a problem containing infinite states or loopy paths

It could gain info from previous iteration

When do we prefer IDS?

When the search space is large and the depth of the solution is not known

* + - * 1. Bidirectional search

How to implement bidirectional search? What’s the advantage of a bidirectional search? And the disadvantage?

Bidirectional search is implemented by replacing the goal test with a check to see whether the frontiers of the two search intersect

The advantage is it cuts the time cost from O(b^d) to O(b^{d/2})

where b is breadth, d is the depth of the shallowest solution

The disadvantage is the predecessors of a state may not be calculable which means it’s hard to carry out the backward search

#### Informed search

Which has help of heuristic functions

* + - * 1. Best-first search

A variety of uniform-cost search, using f instead of g to order the priority queue, where f is an evaluation function, a cost estimate

* + - * 1. Greedy best-first search

A form of best-first search with f(n) = h(n), note this algorithm is not optimal and incomplete

* + - * 1. A\* search

A form of best-first search with f(n) = g(n) + h(n)

What kind of condition h must satisfy so that f is optimal, i.e. A\* search is optimal?

The tree-search version of A\* is optimal if h(n) is admissible, while the graph-search version is optimal if h(n) is consistent.

* + - * 1. Memory-bounded search

Recursive best-first search

Unfamiliar point. A recursive algorithm that attempts to mimic the operation of standard best-first search, using only linear space

Algorithm Overview

Maintain a value flimit, which is the second least f-value on the frontier — current node has the least f-value

For each successor s of the current node c, define its f-value s.f as max(s.g+s.h, c.f) — if s has been explored before, c.f will be greater than or at least equal to s.g+s.h

Recursively explore the successor with least f-value less than flimit (flimit might be changed since all the successors of the current node are added to the frontier), and update its f-value when the recursion unwinds. Continue this process until the goal is reached or all successor’s f-value are greater than flimit. Return the least f-value

def recursive\_best\_first\_search(problem, h=None):  
 """[Figure 3.26]"""  
 # initialize heuristic function  
 h = memoize(h or problem.h, 'h')  
 def RBFS(problem, node, flimit):  
 """  
 Para:  
 flimit: the f-value of the best alternative path available for any ancestor of the current node.  
 Return:  
 goal node if the goal is reached, otherwise, None  
 the lest f-value of its explored successors.  
 """  
 if problem.goal\_test(node.state):  
 return node, 0 # (The second value is immaterial, if we want to know the actual path cost, the second value is node.path\_cost)  
 successors = node.expand(problem)  
 if len(successors) == 0:  
 return None, infinity  
 for s in successors:  
 # node.f is greater than or at least equal to s.path\_cost + h(s) when s has been explored before  
 s.f = max(s.path\_cost + h(s), node.f)  
 while True:  
 # Order by lowest f value  
 successors.sort(key=lambda x: x.f)  
 # Notice that we don't extract the best successor  
 best = successors[0]  
 # successors have no better f-value than flimit  
 if best.f > flimit:  
 # this best.f will assign to node.f eventually as the recursion unwinds  
 # that is the least f-value of node's children   
 return None, best.f  
 if len(successors) > 1:  
 alternative = successors[1].f  
 else:  
 alternative = infinity  
 # update the explored node's f-value  
 result, best.f = RBFS(problem, best, min(flimit, alternative))  
 if result is not None:  
 # goal reached  
 return result, best.f  
 node = Node(problem.initial)  
 # f is not simply g + h, it's the least g(n\*) + h(n\*) where n\* is the a descendant of node on the frontier  
 node.f = h(node)  
 result, bestf = RBFS(problem, node, infinity)  
 return result

Algorithm Description for the recursive function RBFS:

If the current node meets the goal, return it and its f-value if we desire the least path cost

Calculate the f-value of the successors of the current node by max(g(s) + h(s), node.f), which is either the least f-value of the explored descendants of s, if it has been explored before, or otherwise its evaluation function.

While true

Sort the successors of current node based on f-values

If the successor with least f-value is greater than flimit, return None and the least f-value, which will be recursively assigned to the parent of the current node

Recursively call RBFS on the successor with least f-value, flimit passed as argument is the minimum of current flimit and the least f-value of the successor’s siblings

Update the f-value of the successor, which we just called RBFS on, with the returned f-value

If the RBFS we called returned a node, which means the goal is reached, return the returned node and its f-value

Compared to ordinary recursive DFS

We keep track of a value flimit, which is the second least f-value on the frontier. Note that flimit gets smaller as the search goes deeper down the search tree   
Like ordinary recursive DFS, we expand a node with the least f-value at each iteration.   
Unlike ordinary recursive DFS, as the recursion unwinds, we update the f-value of the node just explored instead of throwing it away.   
Furthermore, since we don’t throw away any node, the condition which checks whether we should return switches from whether all children has been explored to whether the least f-value of its children is greater than flimit.   
Also because the f-value of a node is no longer simply g+h if it was explored before, we assign max(successor.g + successor.h, current\_node.f) to each successor at the beginning of each recursion—although this would deviate from the meaning given below (it’ll be less to the nodes that were explored before but not with the least f-value), but that’ll do the job — albeit it’ll lead to more unnecessary iteration (to restore f-value for those explored before but with f-value greater than the least f-value — FYR, see the f-value definition below), but that’s the best we could do if we cannot use extra space to save f-value for each node

frontier may not obvious as it is in ordinary non-recursive search where there is a container keeping track of the nodes in the frontier, but the meaning still stays the same

f-value is not simply g + h, it's the least g(n\*) + h(n\*) where n\* is a descendant on the frontier of the node

Simplified memory-bounded A\* (SMA\*)

A variant of A\* which stores only limited nodes. When memory is full, it drops (back up to its parent) the worst leaf node to add a new one.

### Q & A

#### How to define a problem?

Unfamiliar point

* + - * 1. Determine the initial state that the agent start in
        2. Describe the possible actions available to the agent
        3. Describe what each action does, AKA transition model

Initial state, actions, and transition model together implicitly define the state space

* + - * 1. Determine the goal test
        2. Define a path cost function

#### What’s the difference between tree search and graph search?

* + - * 1. Graph search is an augmented tree search, which stores the states it has been so as to avoid redundant paths

#### How does an agent learn to search better?

* + - * 1. The method rests on an important concept called the meta-level state space

## Beyond Classical Search

### Concept

#### Sensorless problem / conformant problem

A category of problems in which no percept is given

### Algorithm

#### Beam search

An optimization of best-first search. It only searches a predetermined number of the best moves (according to the evaluation function) at the current level rather than all possible moves. It’s not complete and not optimal. But combining with depth-first search, which result in beam-stack search, it comes complete

* + - * 1. What the difference between beam search and SMA\*?

Beam search limits the number of successors to explore for each node, whereas SMA\* limits the total number of nodes on the frontier

#### Local search

Unfamiliar point. A heuristic method for solving computationally hard optimization problem. These algorithms move from solution to solution in the candidate solutions by applying local changes, until a solution deemed optimal is found or a time bound is elapsed.

Note that local search algorithms don’t return path, they only try to find the optimal solution

* + - * 1. Hill-climbing search

An greedy local search which continually moves towards the peak. It often gets stuck for local maxima, ridges, and plateaux

Random-restart hill climbing

An variant of hill-climbing search, it’s complete because it randomly restart with a new randomly picked initial state when getting stuck.

When is a good hill-climbing search not suitable?

A hill-climbing search with a probability p of success is expected to need 1/p restarts, so when p is too small, i.e. there are many local maxima and plateaux, it’s not suitable anymore

* + - * 1. Simulated annealing

A local search that randomly select a successor and adopt it with probability e^{∆E/T} when it’s bad, where ∆E is the difference between the value of the successor and that of current state (so it’s negative for bad successors), T is given by an annealing schedule function, which slowly decreases over time

Algorithm Overview

def exp\_schedule(k=20, lam=0.005, limit=100):  
 """One possible schedule function for simulated annealing"""  
 return lambda t: (k \* math.exp(-lam \* t) if t < limit else 0)  
  
def simulated\_annealing(problem, schedule=exp\_schedule()):  
 """[Figure 4.5] CAUTION: This differs from the pseudocode as it  
 returns a state instead of a Node."""  
 current = Node(problem.initial)  
 for t in range(sys.maxsize):  
 T = schedule(t)  
 if T == 0:  
 return current.state  
 neighbors = current.expand(problem)  
 if not neighbors:  
 return current.state  
 next = random.choice(neighbors)  
 delta\_e = problem.value(next.state) - problem.value(current.state)  
 if delta\_e > 0 or probability(math.exp(delta\_e / T)):  
 current = next

Algorithm description

For t in time bound

Calculate the temperature T based on t

If T is 0, which means the temperature reaches a low-energy state, return current state

Randomly pick a state in its neighbors, adopt it when it’s a better solution or with probability e^{∆E/T} when it’s bad

When T goes to ∞, simulated annealing is like a random walk around its neighbor. When T goes to 0, it’s like hill climbing

* + - * 1. Local beam search

An algorithm that keeps track of k states rather than just one. It starts with k randomly generated states. At each step, all the successors of all k states constitute a candidate pool. If any one is a goal, the algorithm halts. Otherwise, it selects k best successors from the pool and repeats

Local beam search suffers from a lack of diversity among the k states—it always selects k best successors, an soon it will concentrate on a small region of the state space

What’s the difference between local beam search and a set of random-restart hill climbing running in parallel?

In random-restart search, each search process runs independently of the others, there is no information shared between them. In local beam search, the next k states are selected from the candidate pool, which indicates useful information is passed through the parallel search threads.

* + - * 1. Stochastic beam search

An algorithm alleviating the problem a local beam search has. It chooses k successors at random, with the probability of choosing a given successor being an increasing function of its value, it’s kind of like the process of natural selection.

* + - * 1. Genetic algorithm

A variant of stochastic beam search in which successor states are generated by combining two parent states. In genetic algorithm, each state is represented by a string and is rated by the fitness function. It generates the next k states by following three steps:

1. Randomly select two pairs from the current k states for reproduction, in accordance with the probabilities calculated by the fitness function. Notice that different pairs may contain a same state and some states may just be omitted

2. Create the offspring by crossing over the parent strings at some randomly picked crossover point.

3. Mutation happens to the offspring with a small independent probability

* + - * 1. When are local search algorithms useful?

Local search are used to solve optimization problems and those in no need of finding a path. It’s been widely applied to hard computational problems

Optimization problems is to find the best state according to an objective function, so there is no goal test and no path cost for these problems

* + - * 1. What’s the advantage of local search algorithms?

They use very little memory—usually a constant amount

They can find reasonable solutions in large or infinite (continuous) state space where systematic algorithms are unsuitable

#### And-or search

An search algorithm used in non-deterministic environment, where actions may lead to different results (that’s what the and part deals with)

* + - * 1. def and\_or\_graph\_search(problem):  
            # functions used by and\_or\_search  
            def or\_search(state, problem, path):  
            """returns a plan as a list of actions"""  
            if problem.goal\_test(state):  
            return []  
            if state in path: # run into a cycle, not means the search fails  
            return None  
            for action in problem.actions(state):  
            plan = and\_search(problem.result(state, action),  
            problem, path + [state, ])  
            if plan is not None:  
            return [action, plan]  
            def and\_search(states, problem, path):  
            """Returns plan in form of dictionary where we take action plan[s] if we reach state s."""  
            plan = {}  
            for s in states:  
            plan[s] = or\_search(s, problem, path)  
            if plan[s] is None: # the action leads to a failure state which is intolerable  
            return None  
            return plan  
            # body of and or search  
            return or\_search(problem.initial, problem, [])

#### Incremental belief-state search

A search algorithm for unobservable environment. It starts finding a solution that works for the first physical state in the belief state, and check if it works for the second physical state in the belief state; if not, go back find a different solution for the first one, and so on until find a solution that works for every physical state in the belief state

The main advantage of this search is that it is able to detect failure quickly

#### Online search

Unfamiliar point. General reference to search algorithms that interleave computation and action

* + - * 1. Competitive ratio

The ratio between the path costs of the path an online search agent travels to reach a goal and that of the shortest path

* + - * 1. Online DFS agent

class OnlineDFSAgent:  
 """[Figure 4.21] The abstract class for an OnlineDFSAgent. Override  
 update\_state method to convert percept to state. While initializing  
 the subclass a problem needs to be provided which is an instance of  
 a subclass of the Problem class."""  
 def \_\_init\_\_(self, problem):  
 self.problem = problem  
 # state before a is executed  
 self.s = None  
 # action executed at s  
 self.a = None  
 # {state: action\_list}: action\_list is actions haven't tried in state  
 self.untried = defaultdict(list)  
 # {state: ancestor\_list}: ancestor\_list contains all ancestors of state  
 self.unbacktracked = defaultdict(list)  
 # {(start\_state, action): end\_state} end\_state is reached after execute action at start\_state  
 self.result = {}  
 def \_\_call\_\_(self, percept):  
 # the current state  
 s1 = self.update\_state(percept)  
 if self.problem.goal\_test(s1):  
 # goal reached, no action is needed for s1  
 self.a = None  
 else:  
 if s1 not in self.untried.keys():  
 # back up all actions available for s1  
 self.untried[s1] = self.problem.actions(s1)  
 if self.s is not None:  
 if s1 != self.result[(self.s, self.a)]:  
 self.result[(self.s, self.a)] = s1  
 self.unbacktracked[s1].insert(0, self.s)  
 # no untried action  
 if len(self.untried[s1]) == 0:  
 if len(self.unbacktracked[s1]) == 0:  
 # search ended, goal cannot be reached  
 self.a = None  
 else:  
 # else a <- an action b such that result[s', b] = POP(unbacktracked[s'])  
 unbacktracked\_pop = self.unbacktracked[s1].pop(0) # == self.s  
 for (s, b) in self.result.keys():  
 if self.result[(s, b)] == unbacktracked\_pop:  
 # here require actions reversible  
 self.a = b  
 break  
 else:  
 self.a = self.untried[s1].pop(0)  
 self.s = s1  
 return self.a  
 def update\_state(self, percept):  
 """To be overridden in most cases. The default case  
 assumes the percept to be of type state."""  
 return percept

\_\_call\_\_(percept):

Compute the percept-corresponding state s1

If s1 is the goal, search completes

If s1 haven’t been explored, back up all actions available for s1

If there is any untried action available for s1, do that action

Else do action that backtracks to the state before s1

Goal cannot be reached if there is no state for s1 to backtrack to

* + - * 1. Learning real-time A\* (LRTA\*)

Unfamiliar point. An informed algorithm takes action based on the minimum cost estimates of its neighbors. That is, it constantly moves to a state s1 with minimum c(s, s1) + h(s1), where c is the path cost from s to s1, h is the cost estimate for s1, which is initialized by the heuristic function. After deciding which action is about to take, h(s) is updated to c(s, s1) + h(s1). During this process, h(s) gradually converges to the real path cost from s to the goal state. Because of the action-selection strategy, it doesn’t need to save its ancestors and untried actions as online DFS agent does

It takes O(n^2) to explore an environment of n states.

class LRTAStarAgent:  
 """ [Figure 4.24]  
 Abstract class for LRTA\*-Agent. A problem needs to be  
 provided which is an instanace of a subclass of Problem Class.  
 Takes a OnlineSearchProblem [Figure 4.23] as a problem.  
 """  
 def \_\_init\_\_(self, problem):  
 self.problem = problem  
 # self.result = {} # no need as we are using problem.result  
 # {state, cost\_estimate}: cost\_estimate is the cost estimate to reach the goal from state  
 self.H = {}  
 # state before a is executed  
 self.s = None  
 # action executed at s  
 self.a = None  
 def \_\_call\_\_(self, s1): # as of now s1 is a state rather than a percept  
 if self.problem.goal\_test(s1):  
 self.a = None  
 return self.a  
 else:  
 if s1 not in self.H:  
 self.H[s1] = self.problem.h(s1)  
 if self.s is not None:  
 # self.result[(self.s, self.a)] = s1 # no need as we are using problem.output  
 # minimum cost for action b in problem.actions(s)  
 self.H[self.s] = min(self.LRTA\_cost(self.s, b, self.problem.output(self.s, b),  
 self.H) for b in self.problem.actions(self.s))  
 # an action b in problem.actions(s1) that minimizes costs  
 self.a = argmin(self.problem.actions(s1),  
 key=lambda b:self.LRTA\_cost(s1, b, self.problem.output(s1, b), self.H))  
 self.s = s1  
 return self.a  
 def LRTA\_cost(self, s, a, s1, H):  
 """Returns cost to move from state 's' to state 's1' plus  
 estimated cost to get to goal from s1."""  
 print(s, a, s1)  
 if s1 is None:  
 return self.problem.h(s)  
 else:  
 # sometimes we need to get H[s1] which we haven't yet added to H  
 # to replace this try, except: we can initialize H with values from problem.h  
 try:  
 return self.problem.c(s, a, s1) + H[s1]  
 except:  
 return self.problem.c(s, a, s1) + self.problem.h(s1)

\_\_call\_\_(s1):

If s1 is the goal, search complete

Update h(s) to c(s, s1) + h(s1) if s is not None

Take action which leads to the next state s2 with minimum c(s1, s2) + h(s2) — IMHO, we could omit above update statement and have h(s1) updated here to gain some additional efficiency — update s to s1

* + - * 1. Efficiency comparison between Online DFS and LRTA\*

Online DFS is more efficient in uninformed case, where no heuristics are involved

#### Anytime algorithm

Unfamiliar point. An algorithm that could return a solution even if it is interrupted before it ends. The more time it keeps running, the better solution it's expected to obtain

#### Beam-stack search

An memory-efficient anytime algorithm.

It’s memory complexity is dw, where d is the depth of the search tree and w is the beam width

* + - * 1. Divide-and-Conquer Beam-Stack Search

An optimized beam-stack search, whose memory complexity is only 4w

### Q & A

#### What results in or node? And and node?

* + - * 1. The choice of actions results in or node, while an action leading to different results results in and node

#### What does it mean in a deterministic environment that an action sequence is a solution for a belief state?

* + - * 1. It means after that action sequence is executed, each physical state in that belief states ends up the goal state.

#### How can above conclusion help to improve unobservable search?

Unfamiliar point

* + - * 1. We can discard a path reaching a belief state whose substate has already been generated

If that belief state can reach the goal state with an action sequence, its substate is guaranteed to reach the goal state with the same action sequence or even a shorter one, so there’s no need to consider it anymore.

#### What kind of search algorithm do we use to find a solution in a partially observable environment?

* + - * 1. And-or search

Because the search algorithm is designed before any percept is given, so we need to predict all possible percepts at each belief state and deal with each of them, which is done in the or-part. Because of partial observability, for each possible percept, a belief state contains several physical states and an action sequence is required to cope with each of them. That’s done in the and-part

#### What’s the difference between online search algorithms and search algorithms with observations?

* + - * 1. Online search algorithms make decisions based on the percept history and instant computations, while search algorithms with observations make plans before any percepts received and make decisions in terms of the plan and the current percept

## Adversarial Search

### Terminilogy

#### Ply

In game parlance, a ply refers to one turn taken by one of the players

### Concept

#### Zero-sum game

A game in which each participant’s gain or loss of utility is exactly balanced by the losses or gains of the utility of the other participants

#### Horizon effect

A problem where, in many games, computers can only search a few piles down the game tree. It arise when the program is facing an opponent’s move that causes serious damage and is unavoidable, but program can put it off beyond the depth of searching by trying some moves which are overall detrimental

#### Quiescent search

An algorithm to mitigate horizon effect. It searches deeper and hence delays evaluation until positions are quiescent—that is, unlikely to exhibit wild swings in value in the near future

#### Stochastic game

Games in which chance is involved

#### Monte Carlo simulation

A broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. Or generally speaking, it evaluates an object from experiment

### Algorithm

Iterative deepening are always used with following algorithms in time-limited cases

#### Minimax

An adversary search algorithm, a variant of deep first search

* + - * 1. Algorithm Overview

**def** minimax\_decision(state, game):  
 """Given a state in a game, calculate the best move by searching  
 forward all the way to the terminal states. [Figure 5.3]"""  
 player = game.to\_move(state)  
 **def** max\_value(state):  
 **if** game.terminal\_test(state):  
 **return** game.utility(state, player)  
 v = -float(**'inf'**)  
 **for** a **in** game.actions(state):  
 v = max(v, min\_value(game.result(state, a)))  
 **return** v  
 **def** min\_value(state):  
 **if** game.terminal\_test(state):  
 **return** game.utility(state, player)  
 v = float(**'inf'**)  
 **for** a **in** game.actions(state):  
 v = min(v, max\_value(game.result(state, a)))  
 **return** v  
 # Body of minimax\_decision:  
 **return** argmax(game.actions(state),  
 key=**lambda** a: min\_value(game.result(state, a)))

* + - * 1. How is min-max analogous to and-or algorithm?

Both algorithms are comprised of two interlaced operations

#### Minimax with Alpha-beta pruning

An optimization of minimax algorithm

* + - * 1. Alpha-beta pruning

**def** alphabeta\_search(state, game):  
 """Search game to determine best action; use alpha-beta pruning.  
 As in [Figure 5.7], this version searches all the way to the leaves."""  
 player = game.to\_move(state)  
 # Functions used by alphabeta  
 **def** max\_value(state, alpha, beta):  
 **if** game.terminal\_test(state):  
 **return** game.utility(state, player)  
 v = -infinity  
 **for** a **in** game.actions(state):  
 v = max(v, min\_value(game.result(state, a), alpha, beta))  
 **if** v >= beta:  
 **return** v  
 alpha = max(alpha, v)  
 **return** v  
 **def** min\_value(state, alpha, beta):  
 **if** game.terminal\_test(state):  
 **return** game.utility(state, player)  
 v = infinity  
 **for** a **in** game.actions(state):  
 v = min(v, max\_value(game.result(state, a), alpha, beta))  
 **if** v <= alpha:  
 **return** v  
 beta = min(beta, v)  
 **return** v  
 # Body of alphabeta\_cutoff\_search:  
 best\_score = -infinity  
 beta = infinity  
 best\_action = **None**  
 **for** a **in** game.actions(state):  
 v = min\_value(game.result(state, a), best\_score, beta)  
 **if** v > best\_score:  
 best\_score = v  
 best\_action = a  
 **return** best\_action

* + - * 1. Alpha-beta pruning with evaluation function

**def** alphabeta\_cutoff\_search(state, game, d=4, cutoff\_test=**None**, eval\_fn=**None**):  
 """Search game to determine best action; use alpha-beta pruning.  
 This version cuts off search and uses an evaluation function."""  
 player = game.to\_move(state)  
 # Functions used by alphabeta  
 **def** max\_value(state, alpha, beta, depth):  
 **if** cutoff\_test(state, depth):  
 **return** eval\_fn(state)  
 v = -infinity  
 **for** a **in** game.actions(state):  
 v = max(v, min\_value(game.result(state, a),  
 alpha, beta, depth + 1))  
 **if** v >= beta:  
 **return** v  
 alpha = max(alpha, v)  
 **return** v  
 **def** min\_value(state, alpha, beta, depth):  
 **if** cutoff\_test(state, depth):  
 **return** eval\_fn(state)  
 v = infinity  
 **for** a **in** game.actions(state):  
 v = min(v, max\_value(game.result(state, a),  
 alpha, beta, depth + 1))  
 **if** v <= alpha:  
 **return** v  
 beta = min(beta, v)  
 **return** v  
 # Body of alphabeta\_cutoff\_search starts here:  
 # The default test cuts off at depth d or at a terminal state  
 cutoff\_test = (cutoff\_test **or**  
(**lambda** state, depth: depth > d **or**  
game.terminal\_test(state)))  
 eval\_fn = eval\_fn **or** (**lambda** state: game.utility(state, player))  
 best\_score = -infinity  
 beta = infinity  
 best\_action = **None**  
 **for** a **in** game.actions(state):  
 v = min\_value(game.result(state, a), best\_score, beta, 1)  
 **if** v > best\_score:  
 best\_score = v  
 best\_action = a  
 **return** best\_action

* + - * 1. Cutoff

A condition where the game playing program knows that the position it is considering could not possibly have resulted from best play by both sides and so need not be considered further

* + - * 1. Why some of nodes in min-max tree can be pruned?

In Max decision, we can calculate a temporary max value (Alpha) from the children, Min nodes, examined so far. During the examination of the next Min node, if it runs into a value (cutoff) which is not greater than that temporary max value (Alpha), then we can prune the rest successors: since the Min node won’t have a value greater than Alpha, there is no chance for this node to be chosen by its parent max decision regardless of the outcomes to the rest successors. Similar thing happens in Min decision.

Alpha: The lower bound, also the highest value we have found so far at any choice along the path for Max

Beta: The upper bound, also the lowest value we have found so far at any choice along the path for Min

From above analysis, we can conclude  
1. Alpha is used as a test criterion in Min, and updated in Max   
2. Beta is used as a test criterion in Max, and updated in Min

Alpha and Beta comprise the bound of the value at each decision, i.e., the value at each decision should lie in [Alpha, Beta], otherwise, there is no chance for this node to be chosen by its parent. So once we find out that the value of Min node won’t be larger than Alpha, there is no need to finish the rest of examination and hence we are safe to return. Similar cases for Max node

* + - * 1. How to improve alpha-beta pruning?

Unfamiliar point

Killer heuristic

A technique which attempts to meet cutoff as early as possible, so it can prune more successors and thereby saving time

How to implement killer heuristic?

One way to do so is to use iterative deepening search, so each deepening search can benefit from previous search which provides some indication for the killer moves

Transposition table

In many games, different permutations of the move sequence end up in the same position. A transposition table is used to store the evaluation of the resulting position so as to avoid repeated evaluation

Evaluation function

What makes of a good evaluation function?

The evaluation function should order the terminal states in the same way as the true utility function

The computation must not take too long

The evaluation should be strongly correlated with the actual chance of winning

Forward pruning

A strategy that prunes some moves at a given node immediately without further consideration. Examples include beam search and probabilistic cut.

Beam search. Since alpha-beta pruning is a depth-first search, to gain knowledge of the best moves, we may resort to iterative deepening search

Probabilistic cut.

A technique using statistics gained from prior experience to prune moves less likely to be the best.

It prunes nodes probably outside the range of [alpha, beta]

#### An alternative to iterative deepening: iterative heuristic

Unfamiliar point

* + - * 1. In contrast to iterative deepening growing the search tree one depth at a time, iterative heuristic grows the search tree one step at a time. At each step, a node on the frontier with the best value is chosen to expand, and its successors are added to frontier. Then the best value of the successors back propagate to its ancestors

Sounds familiar? It’s greedy best-first search!

* + - * 1. How to apply iterative heuristic to adversarial search? ([min/max approximation](https://people.csail.mit.edu/rivest/pubs/Riv87c.pdf))

The problem we face here is how to define the best value, since Min and Max nodes have different criteria for the concept of the best value.

Define the generalized p-mean of a  
 then we have its partial derivative with respect to a{i}

The generalized p-mean of a has following properties:

When p approximates ∞, it’s equal to max(**a**)

When p approximate to -∞, it’s equal to min(**a**)

Its partial derivative also has a useful property:

It measures the sensitivity of M to changes in a{i}, since it’s larger for greater a{i}, and smaller for smaller a{i}

With the help of the generalized p-mean of a, we substitute Min and Max as follows.   
For a node c, we define its value:  
 Where **d** is the descendants of c.  
This definition helps unify the concept of best descendant for both Min and Max nodes: the node with greatest ∂v(c)/∂v(d) value is the best.

Above process can only help us to choose the best descendant for a node, but not enough to choose the best node on the frontier, because of the limits of the definition of M — the argument of M is only limited to the descendant of a node.   
To find out the best node on the frontier, we need something which measures all nodes on the frontier equally.  
Let  
 Where x is an ancestor of y.   
D(s, x) is the answer we’re looking for, where s is the root of the search tree and x is a node on the frontier.

According to the definition of partial derivative, D(s, x) measures the sensitivity of v(x) to changes in v(y), which makes it a good answer to our problem

By the chain rule for derivatives  
 Where A(x) is the set of ancestor of x including x itself and f(c) is the parent of c.   
This procedure of calculating D(s, x) suffers the vanishing gradient problem, since D(f(c), c) is a small number. To ease such a problem, we define the path cost or weight on the edge from f(c) to c (alternatively, we could also impose the path cost to node c)  
 So the weight from s to x, where s is the root of the search tree and x is a node on the frontier, is  
 We now can say that the best node on the frontier is the node with the least weight

* + - * 1. Drawback of min/max approximation

At each step, it is not only the path costs of newly added nodes on the frontier that need to be calculated. Expanding a node effects the values of all its ancestors, including the root node s. That means all the path cost needs to be updated at each step, since they all depend on v(s). This could introduce a lot of computational costs

The good news is w(f(c), c) can be rewritten as follows  
 Where d is a sibling of c with the most favorable value v(d), which make v(d) a valid substitution for v(f(c))  
If we approximate log(n) to a small constant number, let’s say 0.05, then we don’t need to recompute all path costs every time. The only nodes that need to be concerned are those whose best sibling has changed due to the new expansion.

* + - * 1. Min/max approximation vs Minimax with Alpha-beta pruning using iterative deepening

Min/max approximation computes less moves, but the computational cost is much higher. When the resource is limited, Minimax with Alpha-beta pruning outperforms Min/max approximation. When the number of moves is limited, the story is reversed

### Q & A

#### How to choose between search and table lookup?

* + - * 1. Table lookup gains advantages at the beginning and the end of the game while search is a better choice in between

#### What kind of condition an evaluation must satisfy for a stochastic game?

* + - * 1. The evaluation function must be a positive linear transformation of the probability of winning from position

#### A normal minimax search takes O(b^m) time to search m levels of a tree with branching factor b, how long does a search take to do the same depth search when each move ends up in one of n different situations, like backgammon?

* + - * 1. O((b n)^m). We don’t consider situation, we just consider opponents moves in different situation, which means the branching faction becomes n times

#### How to search partially observable games?

* + - * 1. Maintain a belief state and do and-or search.
        2. Since we only have partial observation, we generally don’t have enough information to assert that one move is better than the others and hence probability is involved

#### Apart from trying to figure out the best move, what else should we consider in partially observable games?

* + - * 1. Unpredictability, minimizing the information that the opponent has about their location, plays a significant role in partially observable games.

### Further readings

#### [Multi-player alpha-beta pruning](https://www.cc.gatech.edu/~thad/6601-gradAI-fall2015/Korf_Multi-player-Alpha-beta-Pruning.pdf)

* + - * 1. Maxn algorithm

Each node is represented by a tuple of n values. For nodes on the frontier, the value of each component is calculated by an evaluation function. The value of the interior nodes where player i is to move is the tuple of the child for which ith component is a maximum

* + - * 1. Premises for maxn algorithm to use shallow pruning

If there is an upper bound on the sum of all components of a tuple, and there is a lower bound on the values of each component, then actual tree pruning is possible.

An upper bound on the sum of all components is needed so that once we know the lower bound of the player to move at the current node, it is easy to calculate the upper bound for the rest of players at the current node by subtracting the lower bound from the upper bound.

* + - * 1. Immediate pruning

The simplest kind of pruning possible under these assumptions occurs when player i is to move, and the ith component of one of its children equals the upper bound on the sum of all components. In that case, all remaining children can be pruned, since no child's ith component can exceed the upper bound on the sum.

This is equivalent to the situation in two-player case when a child of Max has a value of ∞, or a child of Min has a value of -∞

* + - * 1. Shallow pruning

maxn\_shallow(node, player, upper\_bound, depth)  
 if depth == 0  
 evaluate node and return best = maxn\_shallow(first\_child, next(player), sum, depth - 1)  
 for each remaining child  
 if best[player] > upper\_bound # Since the value of the player to move at the current node exceeds the upper bound, the current node won’t be selected by its parent. The rest of children are no longer needed to consider, thus it’s save to prune them  
 return best current = maxn\_shallow(child, next(player), sum - best[player], depth - 1)  
 if current[player] > best[player] best = current  
 return best

sum is the upper bound on the sum of all components

* + - * 1. Shallow pruning vs alpha-beta pruning

Shallow pruning in maxn algorithm: The upper bound is freshly calculated by subtracting the best value for the current player from the sum of all components

Alpha-beta pruning in minimax algorithm: The upper bound (beta) and the lower bound (alpha) are passed along down the search tree, getting updated in min and max, respectively.

* + - * 1. Why does alpha-beta pruning only work well in the two-player case?

We can view the two-player case in the way that one tries to maximize its own score and the other tries to minimize that score. As a result, It is viable to restrict the score in a range during the search. This is no longer held in the cases with more than two players. In those cases, every player tries to maximize its own score and doesn’t give much consideration on other player’s score. Thus, there won’t be a single range any more

* + - * 1. Conclusion

In the best case, shallow pruning results in signif¡icant savings in computation, but in the average case it does not reduce the asymptotic branching factor.

## Constraint Satisfaction Problems

### Terminology

#### Constraint

* + - * 1. Unary constraint

It restricts the value of a single variable

* + - * 1. Binary constraint

It relates two variables

* + - * 1. Global constraint

It involve arbitrary number of variables

#### Consistency

In classical deductive logic, consistency indicates that there is no contradiction

* + - * 1. Local consistency

It’s the consistency of subset of variables or constraints, including node consistency, arc consistency, and path consistency

Node consistency

A variable in a CSP is node-consistent if all values in the domain of the variable satisfy every unary constraint on that variable

Arc consistency

A variable in a CSP is arc-consistent if every value in its domain is consistent with some admissible value of the second variable in its binary constraints. The most popular algorithm for arc consistency is AC-3.

Path consistency

A two-variable set {Xi, Xj} is path-consistent with respect to Xk if, for every pair of assignment {a, b} consistent with the constraints on {Xi, Xj}, there is an value c in the domain of Xk such that {a, c} and {b, c} satisfy the constraints on {Xi, Xk} and {Xj, Xk}, respectively

K-consistency

A (k-1)-variable set is k consistent with respect to Xk if, for any set of consistent assignment to k-1 variables, there is a consistent assignment to Xk. In particular, 2-consistency is arc consistency. On the other hand, 3-consistency coincides with path consistency only if all constraints are binary

#### No-good

A set of variables, along with their corresponding values that causes inconsistency during backtracking search. Such info can be recorded in form of a new constraint or kept in a separate cashe for constraint learning

#### Cycle cutset

A subset S of the CSP’s variables such that the constraint graph becomes a tree after the removal of S

#### Tree width

The tree width of a tree decomposition is the size of its largest subproblem minus one

#### Symmetry-breaking constraint

Constraints added to reduce the search space by breaking the value symmetry

### Concept

#### Graph

* + - * 1. Constraint graph

In which the nodes correspond to variables while links connect any two variables that participate in a constraint

* + - * 1. Constraint hypergraph

In which nodes correspond either to variables or to constraints and edges exist only between a constraint-node and the corresponding variable-node

* + - * 1. Dual graph

In which nodes represent constraints and two nodes are connected if two constraints share variables

* + - * 1. How to transform an n-ary CSP to a binary one?

Hidden transformation

After the transformation, the original n-ary CSP becomes a constraint hypergraph

Variables: the original variables plus h-variables which represent the original constraints

Domain: the domains of the original variables stay unchanged, the domain of a h-variable consists of a unique identifier for every tuple in the original constraint

Constraint: binary constraint between a original variable and a h-variable iff the original constraint represented by the h-variable involves the original variable

Dual graph transformation

After the transformation, the original n-ary CSP becomes a dual graph

C-variables: constraints in the original CSP

Domain: set of tuples that satisfy the original constraint

Constraint: binary constraint between c-variables iff two constraints share the some variables

#### Linear constraint

In which each variable appears only in linear form

* + - * 1. Linear programming problem

The best known category of continuous-domain CSPs in which each constraints must be equalities and inequalities

What’s the time complexity to solve a linear programming problems?

Polynomial in the number of variables

#### Preference constraint

It prescribes which solution is preferred. It can often be encoded as a cost on individual variable assignment—a less preference with a higher cost

* + - * 1. Constraint optimization problem (COP)

CSPs with preferences constraints

#### Constraint propagation

A specific type of inference using the constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable

### Algorithm

#### AC-3 O(cd^3)

An algorithm to make sure arc consistency. c is the number of constraints, d is the domain size

* + - * 1. def AC3(csp, queue=None, removals=None): # O(cd^3)  
            """  
            Para:  
            queue [(Xi, Xk)]: (Xi, Xk) is a pair of variables in a binary constraint which AC-3 is going to make sure arc consistent  
            removals [(variable, removed\_values)]: assigned variable and its removed\_values during the inference  
            """  
            if queue is None:  
            queue = [(Xi, Xk) for Xi in csp.variables for Xk in csp.neighbors[Xi]]  
            # make sure the csp problem support pruning values from the domain  
            csp.support\_pruning()  
            while queue: # O(cd)  
            (Xi, Xj) = queue.pop()  
            if revise(csp, Xi, Xj, removals):  
            # the domain of Di is revised  
            if not csp.curr\_domains[Xi]:  
            # no value in Xi's domain  
            return False  
            # add to the queue all arcs (Xk, Xi) where Xk is a neighbor of Xi  
            # do so because the change in Di might enable further reductions in the domain of Dk  
            for Xk in csp.neighbors[Xi]:  
            if Xk != Xi:  
            queue.append((Xk, Xi))  
            return True  
           def revise(csp, Xi, Xj, removals):  
            """Reduce Xi’s domain so that it is consistent with Xj, return true if the domain is pruned. """  
            revised = False  
            for x in csp.curr\_domains[Xi][:]: # O(d)  
            # If Xi=x conflicts with Xj=y for every possible y, eliminate Xi=x  
            if all(not csp.constraints(Xi, x, Xj, y) for y in csp.curr\_domains[Xj]): # O(d)  
            csp.prune(Xi, x, removals)  
            revised = True  
            return revised

Algorithm Description

Maintain a queue which contains pairs of variables we want to make sure arc consistency — the first variable is arc consistent with the second

While queue is not empty:

Xi, Xj = queue.pop()

Revise Xi so that it’s arc consistent with Xj

If Xi is revised:

If Xi’s domain is empty:

Arc consistency disobeyed, return false

For every neighbor Xk of Xi

Add (Xk, Xi) to queue

#### Backtracking search

A general algorithm for finding all (or some) solution to some computational problems, notably CSPs, that incrementally builds candidates to the solutions, and abandons each partial candidate c as soon as it determines that c cannot possibly be completed to a valid solution. It’s a variant of DFS, the major difference is the time when it backtracks switch from when all a node’s children have been searched to when a node cannot possibly be completed to a valid solution

* + - * 1. Variable ordering

Minimum-remaining-values (MRV) heuristic

A heuristic to choose the variable with the fewest legal values

def first\_unassigned\_variable(assignment, csp):  
 """The default variable order."""  
 return first([var for var in csp.variables if var not in assignment])  
  
def mrv(assignment, csp):  
 """Minimum-remaining-values heuristic."""  
 # argmin\_random\_tie return a minimum element of a sequence; break ties at random.  
 return argmin\_random\_tie(  
 [v for v in csp.variables if v not in assignment],  
 key=lambda var: num\_legal\_values(csp, var, assignment))  
  
def num\_legal\_values(csp, var, assignment):  
 if csp.curr\_domains:  
 return len(csp.curr\_domains[var])  
 else:  
 # calculate the number of values in var’s domain that don’t conflict with the assignment  
 return count(csp.nconflicts(var, val, assignment) == 0  
 for val in csp.domains[var])

Degree heuristic

A heuristic attempting to reduce the branching factor on future choice by selecting the variable that is involved in the largest number of constraints on other unassigned variable

* + - * 1. Value ordering

Least-constraining-value (LCV) heuristic

A heuristic preferring the value that rules out the fewest choice for the neighboring variables in the constraint graph

def unordered\_domain\_values(var, assignment, csp):  
 """The default value order."""  
 return csp.choices(var)  
  
def lcv(var, assignment, csp):  
 """Least-constraining-values heuristic."""  
 def n\_conflicts(var, val):  
 return np.sum([[csp.constraints(var, val, var2, val2) for val2 in csp.domains[var2]]  
 for var2 in csp.neighbors[var] if var2 not in assignment])  
 return sorted(csp.choices(var),   
 key=lambda val: n\_conflicts(var, val))

* + - * 1. Inference

Forward checking

A technique pruning from the neighboring variables’ domains any value inconsistent with the assigned variable. This process establishes arc consistency for the assigned variable

def forward\_checking(csp, var, value, assignment, removals):  
 """Prune neighbor values inconsistent with var=value.  
 Para:  
 assignment {variable: value}: assigned variable and its corresponding value  
 removals [(variable, removed\_values)]: assigned variable and its removed\_values during the inference  
 """  
 for B in csp.neighbors[var]:  
 if B not in assignment:  
 for b in csp.curr\_domains[B][:]:  
 if not csp.constraints(var, value, B, b):  
 csp.prune(B, b, removals)  
 if not csp.curr\_domains[B]:  
 return False  
 return True

Maintaining Arc Consistency (MAC)

An technique augmenting forward checking, which calls AC-3 to make sure arc consistency is maintained

def mac(csp, var, value, assignment, removals):  
 """Maintain arc consistency.  
 Para:  
 assignment {variable: value}: assigned variable and its corresponding value  
 removals [(variable, removed\_values)]: assigned variable and its removed\_values during the inference  
 """  
 return AC3(csp, [(X, var) for X in csp.neighbors[var]], removals)

* + - * 1. Algorithm Overview

Since backtracking search can be viewed as a variant of DFS, we can implement it non-recursively by using a stack

def backtracking\_search(csp,  
 select\_unassigned\_variable=first\_unassigned\_variable,  
 order\_domain\_values=unordered\_domain\_values,  
 inference=no\_inference):  
 """[Figure 6.5]"""  
 def backtrack(assignment):  
 if len(assignment) == len(csp.variables):  
 return assignment  
 var = select\_unassigned\_variable(assignment, csp)  
 for value in order\_domain\_values(var, assignment, csp):  
 # check whether var=value conflicts with other assigned variables  
 if 0 == csp.nconflicts(var, value, assignment):  
 # add (var, value) to assignment  
 csp.assign(var, value, assignment)  
 # prune the domain of var except value, return [var, removed\_values]  
 removals = csp.suppose(var, value)  
 if inference(csp, var, value, assignment, removals):  
 # inference succeeds: no domain of unassigned variables is empty  
 result = backtrack(assignment)  
 if result is not None:  
 # find a result  
 return result  
 # restore to what the csp is before var=value  
 csp.restore(removals)  
 csp.unassign(var, assignment)  
 return None  
 result = backtrack({})  
 assert result is None or csp.goal\_test(result)  
 return result

Algorithm Description

Select an unassigned variable using MRV or degree heuristic

Order values using LCV

For each value:

If it’s not conflict with the assignment (this test could be omitted if we apply inference at each step):

Add the variable/value pair to the assignment

Apply forward inference or MAC, if the inference doesn’t result in some empty domain:

Recursively call backtrack, if it finds a result, return the result

If the procedure reaches here, which means current value is not a viable option, we remove it from the assignment and undo the inference

If the procedure reaches here, which means there is no solution consistent with the current assignment, returns None

* + - * 1. Why should variable selection be fail-first, but value selection be fail-last?

Variable ordering is the search tree’s layer ordering. By choosing a variable that’s the most likely to fail first, we lift the corresponding layer nearest to the root so that when pruning happens, it prunes larger parts of the tree earlier

For value ordering, the trick is that we only need one solution; therefore it makes sense to look for the most likely values first. If we wanted to enumerate all solutions rather than just find one, then value ordering would be irrelevant. After all, we need to go through all values in that case

* + - * 1. How to reduce the search space?

Inference

Backjumping

A optimization of backtracking to the most recent assignment in the conflict set, which implies the reason for failure

Conflict set

A conflict set for a variable is a subset of the overall assignment in which each assignment shrinks the variable’s domain

What’s the connection between forward checking and backjumping?

Every branch pruned by backjumping is also pruned by forward checking. Hence, simple backjumping is redundant in a forward-checking algorithm, and even further, in a search that uses stronger consistency checking, such as MAC

Conflict-directed backjumping

A variant of backjumping, which slightly changes the definition of the conflict set. The conflict set is the same as defined in backjumping before backjumping happens, when the algorithm backjump from Xj to Xi, Xi’s conflict set conf(Xi) absorbs conf(Xj), that is

conf(Xi) = conf(Xi) ∪ conf(Xj) - {Xi}

What’s the advantage of conflict-directed backjumping over general backjumping?

Sometimes, we’ll see a variable, V, with an empty conflict set, may still not have viable assignment. This is because a previous assignment shrinks the domain of some variable after V, and thereby makes there is no viable assignment for V and its sequent variables. In that case, general backjumping can do nothing but backtrack one step back, which may introduce some unnecessary searching if the one-step-back variable is not the cause of inconsistency. Conflict-directed backjumping merges conflict sets when backjumping happens, and thus provides a guide for backjumping in such cases.

* + - * 1. Constraint learning

A technique to find the minimum subset of assignments from the conflict set that causes inconsistency, and record(learn) them as a no-good so that next time it encounters such a no-good, it immediately evaluates it as inconsistency without further search

#### Local search algorithms for CSP

* + - * 1. How to use local search algorithms to solve CSPs?

A complete-state formulation: the initial state assigns a value to every variable, and the search algorithm changes the value of one variable at a time until the CSP is solved. Min-conflicts heuristic provides an example

* + - * 1. Min-conflicts algorithm

A specialized stochastic hillclimbing search which randomly selects a conflicted variable and change it to the value that results in the minimum number of conflicts with other variables

def min\_conflicts(csp, max\_steps=100000):  
 """Solve a CSP by stochastic hill climbing on the number of conflicts."""  
 # Generate a complete assignment for all variables (probably with conflicts)  
 csp.current = current = {}  
 for var in csp.variables:  
 val = min\_conflicts\_value(csp, var, current)  
 csp.assign(var, val, current)  
 # Now repeatedly choose a random conflicted variable and change it  
 for i in range(max\_steps):  
 conflicted = csp.conflicted\_vars(current)  
 if not conflicted:  
 return current  
 var = random.choice(conflicted)  
 val = min\_conflicts\_value(csp, var, current)  
 csp.assign(var, val, current)  
 return None  
  
def min\_conflicts\_value(csp, var, current):  
 """Return the value that will give var the least number of conflicts.  
 If there is a tie, choose at random."""  
 return argmin\_random\_tie(csp.domains[var],  
 key=lambda val: csp.nconflicts(var, val, current))

Algorithm Description

give an assignment to all variables

repeatedly choose a conflicted variable

assign it a value which results in the minimum number of conflicts with other variables

iterate until there is no variable in conflict or time’s up

* + - * 1. Constraint weighting

Unfamiliar point. A technique to help concentrate the local search on the important constraints. Each constraint is given a numeric weight, Wi, initially all 1. At each step of the search, the algorithm chooses a variable/value pair to change that will result in the lowest total weight of all violated constraints. The weights are then adjusted by incrementing the weight of each constraint that is violated by the current assignment.

This has two benefits:

1. It adds topography to plateaux, making sure that it is possible to improve from the current state

2. It also, over time, adds weight to the constraints that are proving difficult to solve.

* + - * 1. What’s the advantage of local search for CPSs?

It’s surprisingly effective for many CSPs, and can be used in an online setting when the problem changes (e.g. a week’s airline schedule is rendered to failure because of bad weather, it’s easier to reschedule with a local search starting from the current schedule. A backtracking search with the new set of constraints usually requires much more time and might find a solution with many changes from the current schedule)

#### Tree-structured CSP solver (O(n), where n is the number of variables)

Unfamiliar point

* + - * 1. Directed arc consistency (DAC)

Unfamiliar point. A CSP is defined to be directed arc-consistent under an ordering of variable X1, X2, …, Xn if and only if every value in Xi’s domain is arc-consistent with each adjacent Xj for j > i, but not necessarily vice versa. That’s where directed comes from

* + - * 1. Topological sort

def topological\_sort(X, root):  
 """Returns the topological sort of X starting from the root.  
 Input:  
 X is a list with the nodes of the graph  
 N is the dictionary with the neighbors of each node  
 root denotes the root of the graph.  
 Output:  
 stack is a list with the nodes topologically sorted  
 parents is a dictionary pointing to each node's parent  
 Other:  
 visited shows the state (visited - not visited) of nodes  
 """  
 neighbors = X.neighbors  
 visited = defaultdict(lambda: False)  
 stack = []  
 parents = {}  
 build\_topological(root, None, neighbors, visited, stack, parents)  
 return stack, parents  
# Following procedure is essentially a DFS, we have to implement the topological sort in this way so that two variables in a binary constraint are adjacent to each other  
def build\_topological(node, parent, neighbors, visited, stack, parents):  
 """Builds the topological sort and the parents of each node in the graph"""  
 visited[node] = True  
 for n in neighbors[node]:  
 if(not visited[n]):  
 build\_topological(n, node, neighbors, visited, stack, parents)  
 parents[node] = parent  
 stack.insert(0, node)

* + - * 1. Make arc consistent

def make\_arc\_consistent(Xj, Xk, csp):  
 """Make arc between parent (Xj) and child (Xk) consistent under the csp's constraints,  
 by removing the possible values of Xj that cause inconsistencies."""  
 #csp.curr\_domains[Xj] = []  
 for val1 in csp.domains[Xj]:  
 keep = False # Keep or remove val1  
 for val2 in csp.domains[Xk]:  
 if csp.constraints(Xj, val1, Xk, val2):  
 # Found a consistent assignment for val1, keep it  
 keep = True  
 break  
   
 if not keep:  
 # Remove val1  
 csp.prune(Xj, val1, None)  
 return csp.curr\_domains[Xj]

* + - * 1. Algorithm Overview

def tree\_csp\_solver(csp):  
 """[Figure 6.11]"""  
 assignment = {}  
 root = csp.variables[0]  
 X, parent = topological\_sort(csp, root)  
 csp.support\_pruning()  
 for Xj in reversed(X[1:]):  
 if not make\_arc\_consistent(parent[Xj], Xj, csp):  
 return None  
 assignment[root] = csp.curr\_domains[root][0]  
 for Xi in X[1:]:  
 assignment[Xi] = assign\_value(parent[Xi], Xi, csp, assignment)  
 if not assignment[Xi]:  
 return None  
 return assignment

Algorithm Description

Topologically sort the variables in csp

For each variable from the last one to the second in topological order (the root is off the table because there is no parent for it)

Prune its parent’s domain so that its parent is arc consistent with it

If fails, which means there is no valid solution, return none

Now the solution could be any valid assignment from the modified domains

### Q & A

#### What a constraint satisfaction problem consists of?

* + - * 1. X: a set of variables  
           D: a set of domains  
           C: a set of constraints that specify allowable combination of values

#### Are there any special solution algorithms for linear constraints on integer variables? And for nonlinear constraints on integer variables?

* + - * 1. Special solution algorithms for linear constraints on integer variables do exist, but not for nonlinear ones

#### Why might we prefer a global constraint rather than a set of binary constraint?

* + - * 1. It’s easier and less error-prone to write the problem description using a global constraint
        2. It’s possible to design special-purpose inference algorithms for global constraints that are not available for a set of more primitive constraints

#### How to solve a CSP?

* + - * 1. Search
        2. Constraint propagation
        3. Constraint propagation may be interwoven with search, or be done as a preprocessing step, before search start. Sometimes, it can solve the whole problems without search interferes.

#### How to generalize arc consistency to an n-ary constraint?

* + - * 1. A variable Xi is generalized arc consistent with an n-ary constraint, if, for every value v in its domain, there exist a tuple of values that is a member of the constraint, has all its values taken from the domains of the corresponding variables, and is consistent with v

#### What’s the similarity and difference between (1) {Xi, Xj} is path-consistent with Xk and (2) Xi is arc-consistent with Xj, Xi is arc-consistent with Xk, and Xj is arc-consistent with Xk?

* + - * 1. It’s easy to see the difference if we explain (2) as follows:   
           For every pair of assignment {a, b} consistent with the constraint on {Xi, Xj} (Xi is arc-consistent with Xj, this one may not be accurate, but could make do), there is a value c in the domain of Xk such that {a, c} satisfies the constraints on {Xi, Xk} (Xi is arc-consistent with Xk), and there is a value d in the domain of Xk such that {b, d} satisfies the constraints on {Xj, Xk} (Xj is arc-consistent with Xk)  
           In the explanation above, c and d could be different values in the domain of Xk. Only when c is equal to d, (2) is equivalent to (1)

That’s why AC-3 is not sufficient for solving path consistency. It’s sufficient to solve (2), but still too lax to solve path consistency

#### What’s the difference between forward checking and AC-3?

* + - * 1. Forward checking assigns a value to a variable and prunes values in its neighbor so as to make them arc consistent with this variable
        2. For each iteration of the queue in AC-3, it reduces a variable’s domain so that, for each value in the remaining domain, there exists an assignment to its neighbor arc consistent with it

#### How to structure a CSP so as to improve efficiency?

Unfamiliar point

* + - * 1. Divide a it into a set of independent subproblem
        2. Structure it in a tree-structured CSP

How to reduce a general constraint graph to a tree?

Choose a cycle cutset S so that the remaining variables form a tree, now we have a tree for each possible assignment to S and the solution is straightforward: iterate through each possible assignment to S until one’s corresponding tree has a solution

There’s some pre-process to do:

The assignment to S must satisfy all constraints on S and any values inconsistent with the assignment for S should be removed from the domains of the remaining variables

What’s the time complex to find the smallest cycle cutset?

Finding the smallest cycle cutset is NP-hard, but several efficient approximation algorithms are known. The overall algorithmic approach is called cutset conditioning which is introduced in Chapter 14: probabilistic reasoning

Collapse nodes together to form subproblems so that all subproblems form a tree whose constraint is the subproblem solutions agreeing on their shared variable. This is called a tree decomposition. Now all we need to do is to solve subproblems individually and then solve the tree

There are some requirements a tree decomposition must satisfy:

1. Every variable in the original problem appears in at least one of the subproblems

2. If two variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the subproblems

3. If a variable appears in two subproblems in the tree, it must appear in every subproblem along the path connecting those subproblems

1 and 2 ensure that all the variables and constraints are represented in the decomposition

3 reflects the constraint that any given variable must have the same value in every subproblems in which it appears

How to choose a tree decomposition?

To make the tree width as small as possible. Finding the decomposition with minimal tree width is NP-hard, but there are heuristic methods that work well in practice

How to solve a tree-structured CSP?

Order the variables in a topological ordering (By using DFS)

Do AC-3 with a little modification:  
1. We revise variables in the reversed topological ordering, starting from making parent[Xk] consistent with Xk, where Xk is the last variable in topological order  
2. We don’t append any constraint to the queue after a domain is revised, so the time complex is reduced to O(cd^2)

So far, we have made this graph directed arc-consistent.

Modification 1 makes sure that any valid assignment in the topological ordering solve the problem, based on the observation that, once we make every value in parent[Xi]’s domain arc consistent with Xi, as long as Xi’s domain doesn’t shrink, the directed arc consistency is maintained.

Modification 2 also do the same thing, the above observation illustrates the importance of revise ordering to directed arc consistency, so we don’t want to append any constraint to the queue to mess up the reversed topological ordering.

Now any valid assignment in the topological ordering solve the problem

### Further readings

#### On the Conversion between Non-Binary and Binary CSPs

Unfamiliar point

* + - * 1. Q & A

How to generalize forward checking to k-consistency?

If an assignment to a variable causes some constraint forward checkable (if not just simply continue the algorithm), check whether the values in the domain of the remaining unassigned variable, along with the values of the assigned variables, satisfies the constraint and prune those inconsistent. If this procedure causes the domain of the unassigned variable empty, then backtrack.

A k-ary constraint, k ≥ 2, is forward checkable if k-1 of its variables has been assigned and the remaining variable is unassigned

When to choose convert a non-binary CSP to a binary one?

For most problems the non-binary representation is the most efficient representation.

There is a class of problems, when the constraints are tight (which means the domains of the constraints are relatively small) and restrictive (which means that there are large proportion of variables involved in each constraint), for which the binary translations can be more efficient by orders of magnitude

## Logical Agents

### Terminology

#### Knowledge base

A set of sentences representing the facts about the world

#### Model

A mathematical abstraction of possible world which fixes the truth or falsehood of every relevant sentence. We say a modal m satisfies a sentence a or m is a model of a, if a is true in m. We use the notation M(a) to mean the set of all model of a

#### Backus–Naur form (BNF)

A notation technique for context-free grammars, often used to describe the syntax of languages used in computing. Check [BNF in Wiki](https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form) for examples

#### Complex sentence

A sentence constructed from simpler sentence, using parentheses and logical connectives.

* + - * 1. Atomic sentence

A sentence consisting of a single propositional symbol

Propositional symbol

A symbol standing for a proposition that can be true or false

What’s the difference between symbols and literals?

A literal either is a symbol or a negation of a symbol

* + - * 1. Logical connectives

A symbol used to connect two or more sentences, including not(¬), and(∧), or(∨), implies(⇒), iff(⇔)

Negation

Conjunction

Conjunct

Parts of conjunction

Disjunction

Disjunct

Parts of disjunction

Implication / rule

Biconditional

#### Clause

A disjunction of literals

* + - * 1. Unit clause

A clause with just one literal

#### Underconstrained

An underconstrianed problem is one with relatively few constraints on the variables (m/n is small, where m is the number of constraints, n is the number of variables) and thus, it has many solutions. In SAT, it’s the one with relatively few clauses constraining the variables

#### Overconstrained

An overconstrained problem is one with relatively many constraints on the variables and thus it is likely to have no solution

#### Fluent

A synonym for "state variable", representing an aspect of the world that changes

#### Atemporal variable

A symbol associated with a permanent aspect of the world

#### Complementary literals

Two literals are complementary literals if one is the negation of the other

### Concept

#### Knowledge-based system

A computer program that reasons and uses a knowledge base to solve complex problems. It has three types of subsystem: a knowledge base, an user interface, and an inference engine

#### Knowledge

* + - * 1. Procedural knowledge / imperative knowledge

The knowledge about how, and especially how best, to perform some task.

* + - * 1. Declarative knowledge / descriptive knowledge / propositional knowledge

The knowledge about something.

* + - * 1. What’s the difference between procedural knowledge and declarative knowledge.

Procedural knowledge can be regarded as implementations, thus it’s domain-dependent and less general but more detailed

Declarative knowledge can be regarded as ideas, thus it’s more abstract and flexible

Examples:   
1. The knowledge about what an algorithm does is declarative knowledge  
2. The knowledge about the implementation of that algorithm is procedural knowledge

#### Entailment / Logical consequence

A concept in logic describing the relationship between statements that holds true when one statement b logically follows (or is entailed from) one or more statements a (a ⊨ b), which means in every model in which a is true, b is also true

#### Model checking

An inference algorithm which checks the relationship between M(a) and M(b) so as to decide whether one entails another.

* + - * 1. How does model checking infer that a entails b by checking the relationship between M(a) and M(b)?

If M(a) ⊆ M(b), a entails b

#### Properties of a logical system

* + - * 1. Soundness

A logical system is sound if it only derives things that make sense (make no contradictions)

* + - * 1. Completeness

A logical system is complete if it can derive everything that logically follows given statements

* + - * 1. What’s the result of a logical system if it’s sound but not complete, and if it’s unsound but complete?

A sound but not complete one draws true answers but may not all of them

A unsound but complete one draws false answers as well as all true answers

so you can’t make sure the answers it draws are all right

#### Grounding

The connection between logical reasoning processes and the real environment in which the agent exists

#### Sentence

* + - * 1. Properties

Validity

A sentence is valid if it’s true in all models, such a sentence is called a tautology

Satisfiability

A sentence is satisfiable if it’s true in some models

* + - * 1. Conjunctive normal form (CNF)

A sentence is in CNF if it’s a conjunction of clauses, where a clause is a disjunction of literals

* + - * 1. Horn clause

A disjunction of literals of which at most one is positive

Goal clause

A disjunction of literals of which none is positive

The reason we call it goal clause is because, for a horn clause in implication form where P1 ∧ … ∧ Pn → Q, we only need to prove P1 ∧ … ∧ Pn and then Q logically follows. So P1 ∧ … ∧ Pn is our goal to prove. The negation of the goal is thus considered as the goal in clause form and is called goal clause

Definite clause

A disjunction of literals of which exactly one is positive

What makes Horn clauses interesting?

A Horn clause can be written as an implication whose premise is a conjunction of positive literals

e.g.,

¬a ∨ ¬b ∨ c is equivalent to a ∧ b ⇒ c

¬a ∨ ¬b is equivalent to a ∧ b ⇒ false

Inference with Horn clause can be done through the forward-chaining and backward-chaining algorithms

Deciding entailment with Horn clauses can be done in time that is linear in the size of the knowledge base

Forward chaining

A method of reasoning, data-driven

Backward chaining

A method of reasoning, goal-directed

Which algorithm is backward chaining similar to?

And-or graph search.

If there are several rules whose conclusion is what we desire, go through each one in the or-part. In the and-part, iterate over all literals in a rule’s premise.

What’s the time complexity of forward chaining and of backward chaining?

Linear time in the size of the knowledge base

#### Modus Ponens (Latin for model that affirms)

AKA implication elimination, P implies Q and P is asserted to be true, and therefore Q must be true

#### Boolean Satisfiability Problem / Propositional Satisfiability Problem (SAT)

The problem of determining if there exists an interpretation that satisfies a given boolean formula

#### Reductio ad absurdum

A proof technique, literally, "reduction to absurdity"

#### Resolution

A rule of inference leading to refutation theorem-proving technique for sentences in propositional logic and first-order logic (i.e., from a ⊨ b to a ∧ ¬b)

* + - * 1. Resolution closure

A resolution closure RC(S) of a set of clauses S is the set of all clauses derivable by repeated application of the resolution rule to clauses in S or their derivatives

* + - * 1. Describe the procedure of the resolution rule in propositional logic

Given clauses  
C1: l{1} ∨ … ∨ l{k}  
C2: m{1} ∨ … ∨ m{n}  
We could obtain  
C3: l{1] ∨ … ∨ l{i-1} ∨ l{i+1} ∨ … ∨ l{k} ∨ m{1} ∨ … ∨ m{j-1} ∨ m{j+1} ∨ … ∨ m{n}  
If l{i} and m{j} are complementary literals

We say l{i} in C1 resolves with m{j} in C2 to give the resolvent C3

This says that the resolution takes two clauses and produces a new clause containing all the literals of the two original clauses except the two complementary literals

* + - * 1. Why a resolution-based theorem prover can decide whether a ⊨ b?

Every sentence of propositional logic can be expressed in CNF

a ⊨ b is logically equivalent to a ∧ ¬b is unsatisfiable

* + - * 1. How to use the resolution rule to infer a from KB?

Covert KB ∧ ¬a into CNF, construct a set consisting of the resulting clauses

Apply the resolution rule to the resulting clauses, add the new clause to the set if it’s not already present

The process continues until:  
There are no new clauses that can be added, in which KB does not entail a  
Two clauses resolve to yield the empty clause, in which KB entails a

* + - * 1. Why can we infer a from KB, if ¬a and KB resolve to yield the empty clause?

The empty clause—a disjunction of no disjuncts—is equivalent to false because a disjunction is true only if at least one of its disjuncts is true. That implies there is no model for the empty clause, which in turn indicates there is no model for the resolution closure containing the empty clause. KB ∧ ¬a is, therefore, unsatisfiable, we have KB ⊨ a

Similar to a disjunction of no disjuncts, we can regard a conjunction of no conjuncts as true

* + - * 1. What does it mean that resolution is sound? And resolution is complete?

That resolution is sound means what resolution derives is logical

That resolution is complete means resolution derives everything that’s logical. Furthermore, it implies, in proof by contradiction, that clauses in KB ∧ ¬a eventually resolve to yield the empty clause if KB ⊨ a

* + - * 1. How to prove the completeness of resolution?

Unfamiliar point. i.e. prove the ground resolution theorem: if a set of clause is unsatisfiable, then the resolution closure of those clause contains the empty clause

We prove its contrapositive: if the resolution closure RC(S) does not contain the empty clause, then S is satisfiable  
We construct a model for S with suitable truth value for P{1}, …, P{k} as follows  
 For i from 1 to k  
 if a clause C in RC(S) contains the literal ¬P{i} and all C’s other disjuncts are false under the assignment chosen for P{1}, …, P{i-1} (that means all C’s other disjuncts must be assigned by now) P{i} = False  
 else P{i} = True  
To see this construction always produces a model for S, assume the opposite—that, at some stage i in the sequence, assigning P{i} causes some clause C in S to become false (Also we assume that stage i is the first time a falsified clause appears).  
For this to happen, at stage i, C must look like (false ∨ … ∨ false ∨ P{i}) and there must be a clause D that looks like (false ∨ … ∨ false ∨ ¬ P{i}) in RC(S) so that C becomes false at stage i  
Because RC(S) is closed under resolution, RC(S) contains the resolvent of C and D, (false ∨ … ∨ false), which has already falsified before stage i. This contradicts our assumption. Hence, we have proved that this construction always constructs a model of RC(S) and thus a model of S (since S is contained in RC(S)). But if RC(S) contains the empty clause, this construction fails before it begins, because we regard the empty clause as false

It’s important to keep in mind that clauses in the resolution closure are comprised of disjunctions

#### The frame problem

Unfamiliar point. The frame problems arise when time is involved and we only specify which conditions are changed by the actions, but it doesn’t entails that all other conditions remaining unchanged

* + - * 1. How to solve the frame problem?

The frame problem arise because an action doesn’t specify what remains unchanged as the result of an action, we can simply specify the unchanged fluents for each action, such statements are called frame axioms and has schema:  
Action ⇒ (Fluent(t) ⇔ Fluent(t+1))

This solution is remarkably inefficient, since we need to state every irrelevant fluent for each action: in a world with m different actions and n fluents, the set of frame axioms will be of size O(mn). This specific manifestation of the frame problem is sometimes called the representational frame problem

The previous method defines axioms based on actions, we can do it differently in terms of fluents. Such axioms are called successor-state axioms and has schema:  
Fluent(t+1) ⇔ ActionCausesF(t) ∨ (Fluent(t) ∧ ¬ActionCausesNotF(t))

The size of successor-state axiom is of size O(n), which is definitely less than O(mn) in the case of frame axiom

### Algorithm

#### Sentence evaluation

* + - * 1. def pl\_true(exp, model={}):  
            """Return True if the propositional logic expression is true in the model,  
            and False if it is false. If the model does not specify the value for  
            every proposition, this may return None to indicate 'not obvious';  
            this may happen even when the expression is tautological."""  
            if exp in (True, False):  
            return exp  
            op, args = exp.op, exp.args  
            if is\_prop\_symbol(op):  
            # exp is atomic sentence  
            return model.get(exp)  
            elif op == '~':  
            p = pl\_true(args[0], model)  
            if p is None:  
            return None  
            else:  
            return not p  
            elif op == '|':  
            result = False  
            for arg in args:  
            p = pl\_true(arg, model)  
            if p is True:  
            return True  
            if p is None:  
            result = None  
            return result  
            elif op == '&':  
            result = True  
            for arg in args:  
            p = pl\_true(arg, model)  
            if p is False:  
            return False  
            if p is None:  
            result = None  
            return result  
            p, q = args  
            if op == '==>':  
            return pl\_true(~p | q, model)  
            elif op == '<==':  
            return pl\_true(p | ~q, model)  
            pt = pl\_true(p, model)  
            if pt is None:  
            return None  
            qt = pl\_true(q, model)  
            if qt is None:  
            return None  
            if op == '<=>':  
            return pt == qt  
            elif op == '^': # xor or 'not equivalent'  
            return pt != qt  
            else:  
            raise ValueError("illegal operator in logic expression" + str(exp))

#### Convert into CNF

* + - * 1. Eliminate implications

def eliminate\_implications(s):  
 """Change implications into equivalent form with only &, |, and ~ as logical operators."""  
 s = expr(s)  
 if not s.args or is\_symbol(s.op):  
 return s # Atoms are unchanged.  
 args = list(map(eliminate\_implications, s.args))  
 a, b = args[0], args[-1]  
 if s.op == '==>':  
 return ~a | b  
 elif s.op == '<==':  
 return a | ~b  
 elif s.op == '<=>':  
 return (~a | b) & (a | ~b)  
 elif s.op == '^':  
 assert len(args) == 2 # TODO: relax this restriction  
 return (a & ~b) | (~a & b)  
 else:  
 assert s.op in ('&', '|', '~')  
 return Expr(s.op, \*args)

Algorithm Description:

If s is empty or a symbol, do nothing and return s

For each argument in s, recursively call eliminate\_implications on it

Now we’re about to dealing with the resulting s

If the operator is ==>, return ¬a | b

If the operator is <==, return a | ¬b

If the operator is <=>, return (¬a | b) & (a | ¬b)

If the operator is ^, return (a | b) & (¬a | ¬b)

* + - * 1. Move not inwards

def move\_not\_inwards(s):  
 """Rewrite sentence s by moving negation sign inward.  
 >>> move\_not\_inwards(~(A | B))  
 (~A & ~B)"""  
 s = expr(s)  
 if s.op == '~':  
 def NOT(b):  
 return move\_not\_inwards(~b)  
 a = s.args[0]  
 if a.op == '~':  
 return move\_not\_inwards(a.args[0]) # ~~A ==> A  
 if a.op == '&':  
 return associate('|', list(map(NOT, a.args)))  
 if a.op == '|':  
 return associate('&', list(map(NOT, a.args)))  
 return s  
 elif is\_symbol(s.op) or not s.args:  
 return s  
 else:  
 return Expr(s.op, \*list(map(move\_not\_inwards, s.args)))

Algorithm Description:

If s is empty or a symbol, do nothing an return s

If the operator is not ¬, recursively call move\_not\_inwards on each of its arguments

If the operator is ¬, we check the operator of its argument, a

If it’s ¬, recursively call move\_not\_inwards on a’s argument

If it’s &, recursively call move\_not\_inwards on a’s arguments, then associate them by |

If it’s |, recursively call move\_not\_inwards on a’s arguments, then associate them by &

Return s

def associate(op, args):  
 """Given an associative op, return an expression with the same  
 meaning as Expr(op, \*args), but flattened -- that is, with nested  
 instances of the same op promoted to the top level.  
 >>> associate('&', [(A&B),(B|C),(B&C)])  
 (A & B & (B | C) & B & C)  
 >>> associate('|', [A|(B|(C|(A&B)))])  
 (A | B | C | (A & B))  
 """  
 args = dissociate(op, args)  
 if len(args) == 0:  
 return \_op\_identity[op]  
 elif len(args) == 1:  
 return args[0]  
 else:  
 return Expr(op, \*args)  
  
\_op\_identity = {'&': True, '|': False, '+': 0, '\*': 1}  
  
def dissociate(op, args):  
 """Given an associative op, return a flattened list result such  
 that Expr(op, \*result) means the same as Expr(op, \*args)."""  
 result = []  
 def collect(subargs):  
 for arg in subargs:  
 if arg.op == op:  
 collect(arg.args)  
 else:  
 result.append(arg)  
 collect(args)  
 return result

* + - * 1. Distribute and over or

def distribute\_and\_over\_or(s):  
 """Given a sentence s consisting of conjunctions and disjunctions  
 of literals, return an equivalent sentence in CNF.  
 >>> distribute\_and\_over\_or((A & B) | C)  
 ((A | C) & (B | C))  
 """  
 s = expr(s)  
 if s.op == '|':  
 s = associate('|', s.args)  
 if s.op != '|':  
 return distribute\_and\_over\_or(s)  
 if len(s.args) == 0:  
 return False  
 if len(s.args) == 1:  
 return distribute\_and\_over\_or(s.args[0])  
 conj = first(arg for arg in s.args if arg.op == '&')  
 if not conj:  
 return s  
 others = [a for a in s.args if a is not conj]  
 rest = associate('|', others)  
 return associate('&', [distribute\_and\_over\_or(c | rest)  
 for c in conj.args])  
 elif s.op == '&':  
 return associate('&', list(map(distribute\_and\_over\_or, s.args)))  
 else:  
 return s

Algorithm Description:

If the operator is &, recursively call distribute\_and\_over\_or on its arguments, return the flattened result

If the operator is |, get the first conjunction. For each conjunct, associate them with the rest of arguments by |, then recursively call distribute\_and\_over\_or on the resulting disjunction. Associate the results by &, then return

Return s for other cases

* + - * 1. Algorithm overview

def to\_cnf(s):  
 """Convert a propositional logical sentence to conjunctive normal form.  
 That is, to the form ((A | ~B | ...) & (B | C | ...) & ...) [p. 253]  
 >>> to\_cnf('~(B | C)')  
 (~B & ~C)  
 """  
 if isinstance(s, str):  
 s = expr(s)  
 s = eliminate\_implications(s) # eliminate ==>, <=>  
 s = move\_not\_inwards(s) # move ¬ inward  
 return distribute\_and\_over\_or(s) # distribute ∧ over ∨, the resulting expr is in CNF

Algorithm Description:

Eliminate implications and biconditionals

Move ¬ inward

Distribute ∧ over ∨ so that the resulting expression is in CNF

#### Determine entailment by inference

* + - * 1. Determine entailment using the resolution rule

Resolve

def pl\_resolve(ci, cj):  
 """Return all clauses that can be obtained by resolving clauses ci and cj."""  
 clauses = []  
 # we only detect a pair of complementary literals at a time, this procedure may result in many clauses  
 for di in disjuncts(ci):  
 for dj in disjuncts(cj):  
 if di == ~dj or ~di == dj:  
 dnew = unique(removeall(di, disjuncts(ci)) +  
 removeall(dj, disjuncts(cj)))  
 # if dnew is empty, associate will return False, otherwise return a disjunction of Expr  
 clauses.append(associate('|', dnew))  
 return clauses

Algorithm overview

def pl\_resolution(KB, alpha):  
 """Propositional-logic resolution: say if alpha follows from KB. [Figure 7.12]"""  
 clauses = KB.clauses + conjuncts(to\_cnf(~alpha))  
 new = set()  
 while True:  
 n = len(clauses)  
 pairs = [(clauses[i], clauses[j])  
 for i in range(n) for j in range(i+1, n)]  
 for (ci, cj) in pairs:  
 resolvents = pl\_resolve(ci, cj)  
 if False in resolvents:  
 # there is an empty resolvent, KB entails alpha  
 return True  
 new = new.union(set(resolvents))  
 if new.issubset(set(clauses)):  
 # no new clauses added, KB doesn't entail alpha  
 return False  
 for c in new:  
 if c not in clauses:  
 clauses.append(c)

Algorithm Description:

Convert ¬alpha into CNF, of which clauses, along with those of KB, form our clause pool

Resolve each pair of clauses in our clause pool, check the resolvents

If the resolvents contain empty clause, return true

If the resolvents is a subset of our clause pool, return false

Merge the resolvents into our clause pool, and repeat above work

How to further improve the algorithm?

By choosing one clause in alpha, this algorithm can gain additional efficiency, but in that case KB must be a resolution closure. Or we can slightly relax the requirement — that is, KB must be consistent , if we restrict one clause to alpha or its resolvents

* + - * 1. Forward chaining

def pl\_fc\_entails(KB, q):  
 """Use forward chaining to see if a PropDefiniteKB entails symbol q.  
 [Figure 7.15]  
 >>> pl\_fc\_entails(horn\_clauses\_KB, expr('Q'))  
 True  
 """  
 # a dict, where count[c] is the number of symbols in c's premise  
 count = {c: len(conjuncts(c.args[0]))  
 for c in KB.clauses  
 if c.op == '==>'}  
 # a dict, where inferred[s] means whether s has been inferred  
 inferred = defaultdict(bool)  
 # a list, containing all positive symbols in KB  
 agenda = [s for s in KB.clauses if is\_prop\_symbol(s.op)]  
 while agenda:  
 p = agenda.pop()  
 if p == q:  
 return True  
 if not inferred[p]:  
 inferred[p] = True  
 for c in KB.clauses\_with\_premise(p):  
 # one more literal in c's premise is inferred  
 count[c] -= 1  
 if count[c] == 0:  
 # c's premise is met  
 agenda.append(c.args[1])  
 return False

Algorithm Description:

For every implication expression, compute the number of symbols, count, in its premise.

For every not inferred symbol, p, in KB, and every implication in whose premise p appears, decrease the corresponding count by 1. If the count reaches 0, add the conclusion to KB.

Repeat above process until the desired symbol is entailed, for which return true; or all symbol in KB is inferred, for which return false

#### Determine entailment by enumerating the truth table

* + - * 1. def tt\_entails(kb, alpha):  
            """Does kb entail the sentence alpha? Use truth tables. For propositional  
            kb's and sentences. [Figure 7.10]. Note that the 'kb' should be an  
            Expr which is a conjunction of clauses.  
            >>> tt\_entails(expr('P & Q'), expr('Q'))  
            True  
            """  
            assert not variables(alpha)  
            return tt\_check\_all(kb, alpha, prop\_symbols(kb & alpha), {})  
             
           def tt\_check\_all(kb, alpha, symbols, model):  
            """Auxiliary routine to implement tt\_entails."""  
            if not symbols:  
            if pl\_true(kb, model):  
            # model satisfies kb, return the result whether model satisfies alpha  
            result = pl\_true(alpha, model)  
            assert result in (True, False)  
            return result  
            else:  
            # model doesn't satisfy kb, so kb -> alpha is always true  
            return True  
            else:  
            # separate one propositional symbol and enumerate its truth value  
            P, rest = symbols[0], symbols[1:]  
            # following `and` suggests whenever a model satisfies kb, it satisfies alpha, if not, return false   
            return (tt\_check\_all(kb, alpha, rest, extend(model, P, True)) and  
            tt\_check\_all(kb, alpha, rest, extend(model, P, False)))

Algorithm Description:

Recursively enumerate all the models. For those which satisfy KB but not alpha, return false, return true otherwise.

#### SAT solver

SAT solvers are useful to make a plan. Furthermore, those which can tell if the conclusion is unsatisfiable can be used to determine entailment.

* + - * 1. DPLL

A backtracking-based search algorithm for deciding the satisfiability of propositional logic formulae in CNF, i.e. for solving the CNF-SAT problem

Find pure symbol

Which appears only as a positive literal (or only as a negative literal) in any clause

def find\_pure\_symbol(symbols, clauses):  
 """Find a symbol and its value if it appears only as a positive literal  
 (or only as a negative) in clauses.  
 >>> find\_pure\_symbol([A, B, C], [A|~B,~B|~C,C|A])  
 (A, True)  
 """  
 for s in symbols:  
 found\_pos, found\_neg = False, False  
 for c in clauses:  
 if not found\_pos and s in disjuncts(c):  
 found\_pos = True  
 if not found\_neg and ~s in disjuncts(c):  
 found\_neg = True  
 if found\_pos != found\_neg:  
 return s, found\_pos  
 return None, None

Algorithm Description

search through clauses for each symbol. If some appears only as a positive literal or only as a negative literal, return it.

Find unit clause

In which only one variable is unbound

def find\_unit\_clause(clauses, model):  
 """Find a forced assignment if possible from a clause with only 1  
 variable not bound in the model.  
 >>> find\_unit\_clause([A|B|C, B|~C, ~A|~B], {A:True})  
 (B, False)  
 """  
 for clause in clauses:  
 P, value = unit\_clause\_assign(clause, model)  
 if P:  
 return P, value  
 return None, None  
  
def unit\_clause\_assign(clause, model):  
 """Return a single variable/value pair that makes clause true in  
 the model, if possible.  
 >>> unit\_clause\_assign(A|B|C, {A:True})  
 (None, None)  
 >>> unit\_clause\_assign(B|~C, {A:True})  
 (None, None)  
 >>> unit\_clause\_assign(~A|~B, {A:True})  
 (B, False)  
 """  
 P, value = None, None  
 for literal in disjuncts(clause):  
 # we assign the symbol in literal to make literal true  
 sym, positive = inspect\_literal(literal)  
 if sym in model:  
 if model[sym] == positive:  
 # model has already made literal true, which in turn has made clause true  
 return None, None  
 # model has made literal false, skip to the next literal  
 elif P:  
 return None, None # more than 1 unbound variable  
 else:  
 P, value = sym, positive # first unbound variable found  
   
 return P, value  
  
def inspect\_literal(literal):  
 """The symbol in this literal, and the value it should take to  
 make the literal true.  
 >>> inspect\_literal(P)  
 (P, True)  
 >>> inspect\_literal(~P)  
 (P, False)  
 """  
 if literal.op == '~':  
 return literal.args[0], False  
 else:  
 return literal, True

Algorithm Description

for clause in clauses

for literal in clause

assign the symbol to make literal true

if model has the same assignment to symbol as above step

skip to next clause

else if we have an retained assignment

skip to next clause

else

retain the assignment for future check

// since clause has unknown truth value, reaching here means clause is a unit clause and the assignment is the only way to make clause true

return the assignment

Algorithm Overview

def dpll\_satisfiable(s):  
 """Check satisfiability of a propositional sentence.  
 This differs from the book code in two ways: (1) it returns a model  
 rather than True when it succeeds; this is more useful. (2) The  
 function find\_pure\_symbol is passed a list of unknown clauses, rather  
 than a list of all clauses and the model; this is more efficient."""  
 clauses = conjuncts(to\_cnf(s))  
 symbols = list(prop\_symbols(s))  
 return dpll(clauses, symbols, {})  
  
def dpll(clauses, symbols, model):  
 """See if the clauses are true in a partial model."""  
 unknown\_clauses = [] # clauses with an unknown truth value  
 for c in clauses:  
 val = pl\_true(c, model)  
 if val is False:  
 # early termination: model doesn't satisfy the propositional sentence  
 return False  
 if val is not True:  
 unknown\_clauses.append(c)  
 if not unknown\_clauses:  
 return model  
 # pure symbol heuristic  
 P, value = find\_pure\_symbol(symbols, unknown\_clauses)  
 if P:  
 return dpll(clauses, removeall(P, symbols), extend(model, P, value))  
 # unit clause heuristic  
 P, value = find\_unit\_clause(clauses, model)  
 if P:  
 return dpll(clauses, removeall(P, symbols), extend(model, P, value))  
 if not symbols:  
 # no symbol is undetermined but DPLL still haven't return, raise an exception  
 raise TypeError("Argument should be of the type Expr.")  
 P, symbols = symbols[0], symbols[1:]  
 return (dpll(clauses, symbols, extend(model, P, True)) or  
 dpll(clauses, symbols, extend(model, P, False)))

Algorithm Description:

Extract clauses whose truth value is unknown in the model (meanwhile, if the model doesn’t satisfy some clause, return false immediately). If there is no such clauses, return the model.

// following steps always assign a value to a symbol, extend the model, and recursively call dpll, for simplicity, we only specify the assignment part, implicitly considering the other two steps follows

If there is a pure symbol in the extracted clauses

assign value to that symbol to make its literal true.

Else if there is a unit clause in the extracted clauses

make that clause true

Else randomly pick a symbol, assign true to it, if it turns out that doesn’t work, then assign false to it

What improvements does DPLL embody over the simple truth-table enumeration algorithm?

Early termination:   
Early termination is possible based on two observations in CNF:  
1. A clause is true if any literal is true (the property of or)  
2. A sentence is false if any clause is false (the property of and)

This is an advantage which we gain from finding a satisfiable assignment instead of determining entailment

Pure symbol heuristic:   
If a sentence has a model, then it has a model with the pure symbols assigned so as to make their literals true

A pure symbol is a symbol that always appears with the same "sign" in all clause

Note that, in determining the purity of a symbol, the algorithm ignore clauses that are already known to be true in the model constructed so far

Unit clause heuristic:  
In the model construction, the symbols in unit clauses is assigned so as to make the unit clauses true

Note that, the unite clause defined here is slightly different from defined in the Terminology. The algorithm determine a clause a unit clause if all literals but one in it are already assigned false in the model constructed so far

How to further improve the performance of DPLL?

Unfamiliar point.

Component analysis:  
Separate unknown\_clauses into disjoint sets (components) that share no unassigned variables. A solver can divide the original problem into several independent smaller problems and thus gain considerable speed.

In a smaller problem, a solver only need to consider the clauses in its own component. Since all smaller problems are independent of each other, the solver simply combines the resulting models without any further consideration

Variable and value ordering:  
In the last part of DPLL, it selects the first symbol in symbols (which can be regarded as randomly select a symbol). We can improve this by choosing the variable that appears most frequently over all remaining clauses, which is exactly what degree heuristic suggests

Intelligent backtracking:  
Backjump to the relevant point of conflict instead of chronological backtracking and use some form of conflict clause learning to record conflicts so as to avoid repeat later in the search

Random restarts:  
If a run appears not to be making progress, we can start over from the top of the search tree and try something different (in variable and value selection). Conflict clauses learned before are retained so as to help prune the search space. Restarting doesn’t guarantee that a solution will be found faster, but it does reduce the variance on the time to solution

Clever indexing:  
The methods to find a pure symbol and to find a unit clause require fast indexing of such things as "the set of clauses in which variable X appears as a positive literal". Speeding up such indexing helps to improve the performance

* + - * 1. WalkSAT

Unfamiliar point. A particularly useful local search algorithm to solve SAT.

This algorithm chooses randomly between two ways to pick which symbol to flip:

1. A symbol minimizing the number of unsatisfied clauses in the new state

2. A random symbol

Algorithm Overview

def WalkSAT(clauses, p=0.5, max\_flips=10000):  
 """Checks for satisfiability of all clauses by randomly flipping values of variables  
 """  
 # set of all symbols in all clauses  
 symbols = set(sym for clause in clauses for sym in prop\_symbols(clause))  
 # model is a random assignment of true/false to the symbols in clauses  
 model = {s: random.choice([True, False]) for s in symbols}  
 for i in range(max\_flips):  
 satisfied, unsatisfied = [], []  
 # divide clauses into two different list: satisfied and unsatisfied  
 for c in clauses:  
 (satisfied if pl\_true(c, model) else unsatisfied).append(c)  
 if not unsatisfied: # if model satisfies all the clauses  
 return model  
 clause = random.choice(unsatisfied)  
 if probability(p):  
 # randomly choose a symbol to flip  
 sym = random.choice(prop\_symbols(clause))  
 else:  
 # choose the symbol in clause that maximizes number of satisfied clauses  
 def sat\_count(sym):  
 # return the the number of clauses satisfied after flipping the symbol.  
 model[sym] = not model[sym]  
 count = len([c for c in clauses if pl\_true(c, model)])  
 model[sym] = not model[sym]  
 return count  
 sym = argmax(prop\_symbols(clause), key=sat\_count)  
 model[sym] = not model[sym]  
 # if no solution is found within the flip limit, we return failure  
 return None

What’s the limitations of WalkSAT?

WalkSAT has been proven particular useful in solving SAT. But WalkSAT cannot detect unsatisfiability—When WalkSAT returns false, we don’t know whether it’s for the input sentence is unsatisfiable or just time runs out. That indicates it cannot be used to determine entailment

* + - * 1. What’s the advantage/disadvantage of WalkSAT compared to DPLL?

WalkSAT is much more efficient to find a model for a sentence

Unlike DPLL, WalkSAT cannot be used to determine entailment because of its limitations

### Q & A

#### Describe the relationship between forward search, backward search, satisfiability algorithms used in entailment, and resolution algorithm

* + - * 1. Forward search starts with the knowledge base
        2. Backward search starts with the query clause
        3. Satisfiability algorithms and resolution algorithm assume the query clause is false, and thereby infer if it’s against the knowledge base. In such procedures, the query could serve as a guide if the knowledge base is known to be consistent

#### Is propositional logic declarative knowledge or procedural knowledge?

* + - * 1. It’s declarative knowledge. For a specific use, the knowledge need to be further explored

#### What’s the difference between declarative, procedural, and functional programming?

* + - * 1. Declarative programming describes a result and get it via a black box, the opposite of procedural / imperative programming. Examples include SQL, regular expressions, and so on
        2. Procedural / imperative programming describes the algorithm and processes the steps, at various degree of abstraction. Examples include all major languages like C and its legacy languages
        3. Functional programming emphasizes the application of functions without side effects and without mutable states. This implies that, for given inputs, it always yields the same output

#### What’s the difference between a ⊢ b and a ⊨ b?

Unfamiliar point

* + - * 1. a ⊢ b means b is provable from a
        2. a ⊨ b means b logically follows a

#### How to use a ⊢ b and a ⊨ b to explain soundness and completeness

* + - * 1. Soundness: Given a ⊢ b then a ⊨ b

If a sound logical system can derive b from a, then b logically follows a

* + - * 1. Completeness: Given a ⊨ b then a ⊢ b

If b logically follows a, then a complete logical system can derive b from a.

#### What’s the worst-case complex of an inference algorithm for propositional logic?

* + - * 1. Every known inference algorithm for propositional logic has a worst-case complex that is exponential in the size of the input

#### What’s the connection between a ⊨ b and a ⇒ b?

* + - * 1. a ⊨ b if and only if a ⇒ b is valid

#### a ⊨ b if and only if what is unsatisfiable?

The what here is a complex sentence

* + - * 1. a ∧ ¬b

So to proof a ⊨ b, we could assume b to be false and show this leads to a contradiction with known axiom a. This proof technique is called proof by refutation or proof by contradiction

#### What is the satisfiability threshold conjecture?

Unfamiliar point

* + - * 1. For a sentence in CNF with m clauses—each clause contains k literals—and n symbols, let r = m/n  
           There is a threshold ratio r{k}, for every k ≥ 3, so that, as n goes infinity, the probability that the sentence is satisfiable becomes 1 for all values of r below r{k}, and 0 for all values above

It says "as n goes infinity", because the probability that the sentence is satisfiable doesn’t just jump from 1 to 0, there is a "cliff" around the threshold ratio r{k} where it goes down. This cliff gets sharper and sharper as n increases

* + - * 1. What kind of SAT is hard to solve?

For a sentence in CNF with m clauses—each clause contains k literals—and n symbols, let r = m/n  
It’s most difficult to solve an SAT with r around the threshold ratio, an overconstrained one is easier to solve. It’s easiest to solve one underconstrained.

#### How to keep track of the world efficiently without going back into the percept history for each inference?

* + - * 1. Associate propositions with time steps

#### What’s the difference between the requirements for entailment and those for satisfiability?

Unfamiliar point

* + - * 1. For entailment, a sound and complete logical system only derives from what’s known and makes no assumption about the unknown knowledge
        2. For satisfiability, a SAT solver could make up the unknown knowledge so as to obtain a model that satisfy the sentence.
        3. Because of the difference described above, a KB for satisfiability is always more detailed, it must specify additional restrictions on the unknown

#### What's the difference in KB between plans made by SAT solvers and those made by propositional inference?

Unfamiliar point

* + - * 1. Because of the requirements for satisfiability, KB for plans made by SAT solvers must be in detail, it requires additional axioms to make sure the KB is complete

#### How many algorithms is available to check whether a sentence (a) logically follows another (KB)?

* + - * 1. There are three choices in general:  
           1. We can enumerate the truth table of KB ⇒ a  
           2. Convert KB ∧ ¬a into CNF, then apply the resolution rule  
           3. Convert KB ∧ ¬a into CNF, then call DPLL

WalSAT is not sufficient in this case because of its limitations

* + - * 1. For KB and a in horn form:  
           1. Forward chaining  
           2. Backward chaining

#### How to use logical inference to construct plans that are guaranteed to achieve its goals (SATPlan)?

Unfamiliar point

* + - * 1. Construct a sentence in CNF, including:  
           1. A collection of assertion about the initial state  
           2. The successor-state axioms for all possible actions at each time up to some maximum time t  
           3. The assertion that the goal is achieved at time t

For a SAT solver to work as expected, we may need additional axioms in the sentence:

The precondition axioms: which state that an action occurrence requires the preconditions to be satisfied in form of action ⇒ preconditions

The action exclusion axioms: which states that some actions cannot be executed simultaneously in form of ¬action1 ∨ ¬action2

* + - * 1. Use a SAT solver to find a satisfying model

That's why we need additional axioms, otherwise, the SAT solver would just make up the missing knowledge to obtain a satisfying model.

This also makes SATPlan a good debugging tool for knowledge bases because it reveals places where knowledge is missing

* + - * 1. Extract from the model those variables that represent actions and are assigned to be true. Together they represent a plan to achieve the goals

## First-Order Logic

### Terminology

#### Symbol

* + - * 1. Constant symbol

Which stands for objects

* + - * 1. Predicate symbol

Which stands for relations

* + - * 1. Function symbol

Which stands for functions

* + - * 1. What’s the difference between relations and functions in FOPC?

A relation is a set of tuples of objects that are related. Tuples describe a certain relationship among the objects in it. Relations have their truth value in a model, which stand for they’re satisfied or not

A function takes one or more objects as input and returns another object as output

#### Term

A logical expression that refers to an object. Constant symbols, function symbols, and variables are all terms

* + - * 1. Ground term

A term without variable

#### Sentence

A boolean-valued formula whose truth value can be told

* + - * 1. What does a sentence in FOPC consist of?

A sentence is comprised of equations, predicate symbols, logical connectives, and terms

#### Fact

A sentence whose truth value is true

#### Syntactic sugar

An extension to or abbreviation of the standard syntax that does not change the semantics

#### Interpretation

The process mapping variables to objects in the model

* + - * 1. Extended interpretation

The process mapping quantified variables to objects in the model

### Concept

#### First-order logic / first-order predicate calculus (FOPC)

A collection of formal systems using quantified variables over non-logical objects to representing knowledge

* + - * 1. What’s the domain of a model for FOPC?

It’s the set of objects

* + - * 1. What does a model for FOPC include?

A set of objects and interpretation mapping symbols to objects

* + - * 1. What’s the advantages of FOPL over propositional logic? An the disadvantages?

Advantages:  
There are objects and relations in FOPL  
It’s more concise benefiting from qualifiers

Disadvantages:  
The concise representation comes with price of high computational cost for inference

#### Principle of compositionality

The principle of compositionality is the principle that the meaning of a complex expression is determined by the meanings of its constituent expressions and the rules used to combine them

#### Logic commitment

* + - * 1. Ontological commitment

An ontological commitment refers to what it assumes about the nature of reality, or what exists in the world

What’s the primary difference between propositional and first-order logic?

The primary difference lies in the ontological commitment made by each language  
Propositional logic assumes that there are facts that hold or do not hold in the world  
First-order logic assumes more; namely, that the world consists of objects with certain relations among them that do or do not hold

* + - * 1. Epistemological commitment

An epistemological commitment refers to the possible states of knowledge that it allows with respect to each fact, or what an agent believes about facts

* + - * 1. What’s the ontological commitments and epistemological commitment of propositional logic and first-order logic?

#### Temporal logic

A logic, representing and reasoning about proposition qualified in terms of time

#### Fuzzy logic

A variant of propositional logic, where the truth value of variables may be any real number within [0, 1]

#### Database semantics

Unfamiliar point. A more intuitive less error-prone semantics for FOPC. It assumes

1. Every constant symbol refers to a distinct object (unique-names assumption)

2. Atomic sentences not known to be true are in fact false (closed-word assumption)

3. Each model contains no more domain elements than those named by the constant symbols (domain closure)

* + - * 1. When is database semantics useful?

Database semantics is most useful when we are certain about the identity of all the objects described in the knowledge base and we have all the facts at hand; in other cases, it is quite awkward.

#### Knowledge engineering

Knowledge engineering (KE) refers to all technical, scientific and social aspects involved in building, maintaining and using knowledge-based systems.

* + - * 1. What’s the standard procedure of the knowledge-engineering process?

Identify the task

Assemble the relevant knowledge

Decide on a vocabulary of predicates, functions, and constants

Encode general knowledge about the domain

Encode a description of the specific problem instance

Pose queries to the inference procedure and get answers

Debug the knowledge base

### Q & A

#### Why would we need another language, if knowledge can already be represented by one?

* + - * 1. Although different language can represent the same knowledge, but the ways they represent the knowledge determine how efficient they draw a conclusion. And usually the more succinct language is the one more efficient

#### What should knowledge representation languages be?

Unfamiliar point

* + - * 1. Knowledge representation languages should be declarative, compositional, expressive, context independent, and unambiguous.

#### What’s the difference between axioms and theorems?

* + - * 1. Axioms are what’s obviously true, and they cannot be proven
        2. Theorems are logical consequences of axioms

## Inference in First-Order Logic

### Terminology

#### Store(s)

Store a sentence s into the knowledge base

#### Fetch(q)

Return all unifiers such that the query q unifies with some sentence in the knowledge base

* + - * 1. How to implement Fetch(q)?

The basic strategy is to unify q against every fact in the knowledge base. But there are some improvements we can do:  
1. Predicate indexing: put different kinds of facts in different buckets, which are stored in a hash table. E.g., separate Know(a, b) and Brother(c, d) in two different buckets, so Know(John, x) won’t try to unify with Brother(c, d)  
2. For each buckets described above, we can do further indexing based on the arguments. E.g., separate Know(John, y) and Know(David, Richard) based on the first argument, Know(John, x) won’t try to unify with Know(David, Richard)

#### Pattern matching

The act of checking a given sequence of tokens for the presence of the constituents of some pattern.

In case of forward chaining in FOPC, the sequence of tokens is KB, the patterns could be the premises of rules,

#### Complementary literals in FOPC

First-order literals are complementary if one unifies with the negation of the other

### Concept

#### Skolemization

Replace the existentially qualified variables with a term f(x\_{1}, …, x\_{n}) (Skolem function) whose function symbol f is new. The arguments of the Skolem function are all the universally quantified variables in whose scope the existential quantifier appears

* + - * 1. Existential Instantiation

Replace the existentially qualified variables with a new constant symbol, which shouldn’t appear elsewhere in the knowledge base

* + - * 1. How to choose between Existential Instantiation and Skolemization?

When the existential quantifier is inside a universal quantification, apply Skolemization.

#### Generalized Modus Ponens

For atomic sentences p\_{i}, p’\_{i}, and q, where there is a substitution 𝜃 such that Substitute(𝜃, p'\_{i}) = Substitute(𝜃, p\_{i}), in another word, 𝜃 makes p’\_{i} identical to p\_{i}, for all i,

(p’\_{1} ∧ … ∧ p’\_{n} ∧ (p\_{i} ∧ … ∧ p\_{n} ⇒ q) ⇒ Substitute(𝜃, q)

* + - * 1. Modus Ponens

P implies Q and P is asserted to be true, so therefore Q must be true

#### Datalog

Unfamiliar point.A logic programming language that is restricted to first-order definite clauses with no function symbols

* + - * 1. The absence of function symbols helps avoid the cost of occur check

#### Logic programming

A type of programming paradigm based on logic. Systems should be constructed by facts and rules about some problem domain.

#### Constraint logic programming

A form of constraint programming, in which logic programming is extended to include concepts from constraint satisfaction

#### Nonconstructive proof

Prove the existence of something without providing an example

### Algorithm

#### Unification

Unfamiliar point. The process finding substitutions that make different logical expressions look identical. Such substitution are called unifiers

* + - * 1. Occur check

The process describing, when matching a variable against a complex term, one must check whether the variable itself occurs inside the term; if it does, the match fails because no consistent unifier can be constructed

def occur\_check(var, x, s):  
 """Return true if variable var occurs anywhere in x  
 (or in subst(s, x), if s has a binding for x)."""  
 if var == x:  
 return True  
 elif is\_variable(x) and x in s:  
 return occur\_check(var, s[x], s)  
 elif isinstance(x, Expr):  
 return (occur\_check(var, x.op, s) or  
 occur\_check(var, x.args, s))  
 elif isinstance(x, (list, tuple)):  
 # if x is a list or tuple of terms, check if var occurs inside of any of the iterms  
 return any(e for e in x if occur\_check(var, e, s))  
 else:  
 return False

Algorithm Description

if var is identical to x

return true

else if x is a variable and x is in s

substitute x using s, recursively call occur\_check on it and return

else if x is a complex expression

recursively call occur\_check on the connective and each subexpression

else

return false since var is not in x

* + - * 1. Match the variable against the term

def unify\_var(var, x, s):  
 # to avoid conflict substitution, check if var or x is already in it   
 if var in s:  
 return unify(s[var], x, s)  
 elif x in s:  
 return unify(var, s[x], s)  
 elif occur\_check(var, x, s):  
 return None  
 else:  
 return extend(s, var, x)  
  
def extend(s, var, val):  
 """Copy the substitution s and extend it by setting var to val; return copy."""  
 s2 = s.copy()  
 s2[var] = val  
 return s2

Algorithm Description

if var in s

substitute var using s, recursively call unify on it and return

else if x in s

substitute x using s, recursively call unify on it and return

else if var occurs inside x (occur check)

return none

else

add [var: x] to s and return

* + - * 1. Algorithm overview

def unify(x, y, s={}):  
 """Unify expressions x,y with substitution s; return a substitution that  
 would make x,y equal, or None if x,y can not unify. x and y can be  
 variables (e.g. Expr('x')), constants, lists, or Exprs. [Figure 9.1]"""  
 if s is None:  
 return None  
 elif x == y:  
 return s  
 elif is\_variable(x):  
 return unify\_var(x, y, s)  
 elif is\_variable(y):  
 return unify\_var(y, x, s)  
 elif isinstance(x, Expr) and isinstance(y, Expr):  
 return unify(x.args, y.args, unify(x.op, y.op, s))  
 elif isinstance(x, str) or isinstance(y, str):  
 return None  
 elif issequence(x) and issequence(y) and len(x) == len(y):  
 if not x:  
 return s  
 return unify(x[1:], y[1:], unify(x[0], y[0], s))  
 else:  
 return None  
  
def is\_variable(x):  
 """A variable is an Expr with no args and a lowercase symbol as the op."""  
 return isinstance(x, Expr) and not x.args and x.op[0].islower()

Algorithm Description

if s is none, which means some unification failed before calling unify

return none

else if x is identical to y

return s because there is nothing left to unify

else if x is a variable

call unify\_var on it

else if y is a variable

call unify\_var on it

else if x and y are expression

recursively call unify on each subexpression and each connective

else

return none, since x and y are not unifiable

#### Forward chaining in FOPC(simple version)

Unfamiliar point

* + - * 1. Standardize variables

def standardize\_variables(sentence, dic=None):  
 """Replace all the variables in sentence with new variables."""  
 if dic is None:  
 dic = {}  
 if not isinstance(sentence, Expr):  
 return sentence  
 elif is\_var\_symbol(sentence.op):  
 if sentence in dic:  
 return dic[sentence]  
 else:  
 v = Expr('v\_{}'.format(next(standardize\_variables.counter)))  
 dic[sentence] = v  
 return v  
 else:  
 return Expr(sentence.op,  
 \*[standardize\_variables(a, dic) for a in sentence.args])  
  
standardize\_variables.counter = itertools.count()

* + - * 1. Algorithm overview

def fol\_fc\_ask(KB, alpha):  
 """A simple forward-chaining algorithm. [Figure 9.3]"""  
 """  
 Return:  
 A unifier that makes alpha identical to some sentence in KB or some sentence inferred from KB or False  
 This function may also fail to terminate in some cases  
 """  
 # TODO: Improve efficiency  
 # constant symbols in KB  
 kb\_consts = list({c for clause in KB.clauses for c in constant\_symbols(clause)})  
 def enum\_subst(p):  
 """A generator function yielding possible substitution for every variables in p"""  
 query\_vars = list({v for clause in p for v in variables(clause)})  
 for assignment\_list in itertools.product(kb\_consts, repeat=len(query\_vars)):  
 theta = {x: y for x, y in zip(query\_vars, assignment\_list)}  
 yield theta  
 # check if there is a unifier that makes alpha identical to a fact in KB  
 for q in KB.clauses:  
 phi = unify(q, alpha, {})  
 if phi is not None:  
 yield phi  
 while True:  
 new = []  
 # for each implication in KB  
 for rule in KB.clauses:  
 premises, conclusion = parse\_definite\_clause(standardize\_variables(rule))  
 # pattern matching  
 for theta in enum\_subst(premises):  
 if set(subst(theta, premises)).issubset(set(KB.clauses)):  
 # theta is the unifier that makes premises identical to some facts in KB  
 subst\_conclusion = subst(theta, conclusion)  
 if all([unify(x, subst\_conclusion, {}) is None for x in KB.clauses + new]):  
 # subst\_conclusion is a new fact to KB and new  
 new.append(subst\_conclusion)  
 phi = unify(subst\_conclusion, alpha, {})  
 if phi is not None:  
 # phi is the unifier that makes subst\_conclusion identical to alpha  
 yield phi  
 if not new:  
 break  
 for clause in new:  
 KB.tell(clause)  
 return None

Algorithm Description

if there is any unifier that makes alpha identical to some clause in KB

return the unifier

else

while True

for each rule in KB

if there is a unifier that makes its premises identical to some facts in KB

use the unifier to substitute the variables in its conclusion

add the substituted conclusion to KB if it’s not already in it

if there is a unifier that makes the substituted conclusion identical to alpha

yield the unifier

Note:

This process modifies the KB and returns a unifier only directly related to the variables in alpha

* + - * 1. How to improve the simple forward-chaining algorithm?

Unifier construction: In the pattern-matching part, we find a unifier by enumerating possible substitutions for the premise of a rule (in enum\_subst) and then check if some makes the premises identical to some set of facts in KB. Such a process is inefficient, and we can improve it by constructing the unifier incrementally: We find a valid substitution for a clause in the premise at each step, and put all the substitutions together to form a unifier at the end. This procedure introduces a new knot to untangle: how to order the clauses so that the total cost is minimized? It turns out that finding an optimal ordering is NP-hard, but good heuristic, such as minimum-remaining-values (MRV) heuristic used for CSPs —clauses whose variables have fewer valid values unify first — are available

This unifier construction is similar to the and part of the backward chaining introduced next, except that it calls unify instead of or to incrementally construct the unifier

Rule selection: For the iterations of the while loop, we have following observation: Every new fact inferred on iteration t must be derived from at least one new fact inferred on iteration t-1. So at iteration t, we only need to check rules whose premise includes a conjunct p\_{i} that unifies with a fact p’\_{i} newly inferred at iteration t-1. The pattern-matching step then fixes p\_{i} to match with p’\_{i}, but allows the other conjuncts of the rule to match with any other facts in KB.

* + - * 1. Comparison between forward chaining in FOPC and that in propositional logic

Forward chaining in propositional logic iterates over the facts in KB. When all the premises of a rule have been enumerated, it adds the conclusion to KB

Forward chaining in FOPC iterates over the rules — it tries to unify the premises of a rule with facts in KB at each step. The difference is introduced because it now needs the entire unifier to substitute the variables in conclusion

#### Backward chaining in FOPC

Unfamiliar point

* + - * 1. Algorithm overview

def fol\_bc\_ask(KB, query):  
 """A simple backward-chaining algorithm for first-order logic. [Figure 9.6]  
 KB should be an instance of FolKB, and query an atomic sentence."""  
 return fol\_bc\_or(KB, query, {})  
  
def fol\_bc\_or(KB, goal, theta):  
 for rule in KB.fetch\_rules\_for\_goal(goal):  
 # to avoid different interpretations for the same variable  
 # consider passing a dict as the second argument of standardize\_variables   
 lhs, rhs = parse\_definite\_clause(standardize\_variables(rule))  
 for theta1 in fol\_bc\_and(KB, lhs, unify(rhs, goal, theta)):  
 yield theta1  
  
def fol\_bc\_and(KB, goals, theta):  
 if theta is None:  
 pass  
 elif not goals:  
 yield theta  
 else:  
 first, rest = goals[0], goals[1:]  
 for theta1 in fol\_bc\_or(KB, subst(theta, first), theta):  
 for theta2 in fol\_bc\_and(KB, rest, theta1):  
 yield theta2

Or(KB, goal, theta):

valid\_thetas = []

for rules whose conclusion is unifiable with goal

if the unifier doesn’t conflict with theta

substitute the variables in the premises of the rule using the unifier

valid\_thetas += And(KB, substituted premises of the rule, theta)

return valid\_thetas

And(KB, goals, theta):

if theta is none // which means a conflict happened in the process of unify in fol\_bc\_or, but if the unifier is checked as done in the pseudocode above, this check is not necessary

return

if goals is empty // which implies that theta is the answer we’re looking for

return theta

valid\_thetas = [theta]

for goal in goals

temp\_thetas = valid\_thetas

valid\_thetas = []

for temp\_theta in temp\_thetas

valid\_thetas += Or(KB, goal, temp\_theta)

return valid\_thetas

Note:

1. We regard a fact as a rule without premise.

2. unify returns none if the substitution conflicts with the theta passed as the third argument

3. Both And and Or return a list of valid unifiers

4. This process doesn’t change the KB, and the unifiers it returns contains some substitutions not directly related to the variables in query

* + - * 1. Comparison between the and-part of backward chaining and the [incremental belief-state search](omnioutliner:///open?focus=gwVusVQ8jnd&row=kTiLnFKQ3B7)

For better comparison, let’s assume the incremental belief-state search tries to find all the solutions instead of one. Thus, in process of the incremental belief-state search, we find all the solutions for the first state in the belief state and test them against the rest states, throwing away those not working

In the and-part of backward chaining, we maintain a set of valid thetas starting with only the theta passed in. For each goal, we expand thetas in the valid-thetas set so that goal unifies with some facts in KB, meanwhile, throwing away throw those not working

* + - * 1. What’s the relationship between backward chaining and resolution?

If we restrict one choice of resolution to the query and its resolvents, we will see resolution shares some similarity with backward chaining. (The if-part holds as long as the knowledge base is consistent) In fact backward chaining is a special case of resolution with a particular control strategy to decide which resolution to perform next

### A & Q

#### Can we tell one sentence is not entailed from another, in FOPC?

* + - * 1. No, because a sentence with universal/existential quantifier is innumerable. Thus the question of entailment for first-order logic is semi-decidable—that is, algorithms exist that say yes to every entailed sentence, but no algorithm exists that also says no to every non-entailed sentence

#### What’s the connection between pattern matching and constraint satisfaction?

Unfamiliar point

* + - * 1. From pattern matching to constraint satisfaction, we view each conjunct as a constraint on the variables it contains
        2. From constraint satisfaction to pattern matching, we express every finite-domain CSP as a single definite clause together with some associated ground facts.

Recall that a definite clause in implication form is p\_{1} ∧ … ∧ p\_{n} ⇒ c

Each constraint is represented by a conjunct in the premiss and its corresponding valid assignments

#### What’s the disadvantage of forward chaining and backward chaining, respectively?

* + - * 1. Forward chaining may generate nodes irrelevant to the goal
        2. Backward chaining may recompute some nodes. Such a defect can be correct by memoization—that is, caching solutions to subgoals as they are found and then reusing those solutions when the subgoals recurs

#### How to convert a sentence of FOPC into CNF?

* + - * 1. It’s almost identical to the propositional case, the main difference is we need to eliminate quantifiers now. We eliminate existential quantifiers through Skolemization, and then drop the universal quantifiers

#### How to apply the resolution rule to FOPC?

* + - * 1. Almost the same as it works for propositional logic, with a little modification: it reduces two unifiable literals and substitutes the corresponding variables in the resulting clause

#### When we ask a question like "who kill the cat?", we expect to receive an substitution. Unfortunately, resolution can produce nonconstructive proof in this case—that is, it can assert "∃x Kills(x, the cat)", but fail to answer "who". How to solve such an issue?

E.g.

The goal is: ¬Kills(x, the cat)

Database contains: Kills(Jack, the cat) ∨ Kills(Ben, the cat), Loves(Jack, the cat), ¬Loves(y, the cat) ∨ ¬Kills(y, the cat)

Case 1: Loves(Jack, the cat) resolve with ¬Loves(y, the cat) ∨ ¬Kills(y, the cat), which resolves with Kills(Jack, the cat) ∨ Kills(Ben, the cat). We get Kills(Ben, the cat), and so have answer {x/Ben}

Case 2: ¬Kills(x, the cat) resolve with Kills(Jack, the cat) ∨ Kills(Ben, the cat), which resolves again with ¬Kills(x, the cat) to yield the empty clause. Now, however, we only know ∃x Kills(x, the cat), but don’t know what x is equal to

* + - * 1. Restrict the allowed resolution steps so that the query variables can be bound only once in a given proof; then we need to backtrack over the possible bindings
        2. Add a special answer literal to the negated goal, which becomes, in the example above, ¬Kills(x, the cat) ∨ Answer(x). The resolution now halts when a clause containing only a single answer literal. Now the case 2 yields Answer(Jack) ∨ Answer(Ben), rather than a single answer literal, so resolution continues until it yields Answer(Ben)

#### How to deal with equality in resolution?

Unfamiliar point

* + - * 1. Demodulation

Clauses: x=y, C ∨ L[z], and x is unifiable with z, the unifier is 𝜃  
Result: Substitute(𝜃, C) ∨ L[Substitute(𝜃, y)]

* + - * 1. Paramodulation

Clauses: x = C2 ∨ y, C1 ∨ L[z], and x is unifiable with z, the unifier is 𝜃  
Result: Substitute(𝜃, C1) ∨ Substitute(𝜃, C2) ∨ L[Substitute(𝜃, y)]

The result is also Substitute(𝜃, C1) ∨ L[Substitute(𝜃, C2) ∨ Substitute(𝜃, y)], but because C2 is a predicate symbol, which means it’s not relevant to the sentence L, and thus it's cool to move it out.

#### How to improve resolution in FOPC?

Unfamiliar point

* + - * 1. Unit preference:  
           Prefer to do resolution where one of the sentences is a single literal. Our aim is to produce the empty clause, so it might be a good idea to prefer inferences that produce shorter clause. That’s exactly what this strategy do
        2. Set of support:  
           Elaborately select a set of clauses—set of support, and prefer to resolve some clause in the set of support with others in the knowledge base first. The resolvent is then added into the set of support. If we choose the set of support so that the remainder of the sentences are jointly satisfiable (i.e. consistent), then set-of-support resolution is complete. The negated query is a good initial for the set of support if the KB is consistent.

The Set of Support (SOS) strategy restricts the resolutions which are allowed to occur to just a subset of those possible. Specifically each resolution must involve a clause from a subset called the set of support. The idea is that the set of clauses excluded from the set of support are consistent within themselves, and resolution tries to produce an empty clause, so the path to the solution must involve a resolution with one of the clauses in the SOS. This has an analogy with a heuristic we teach children to help them with problem solving: if you haven’t used everything mentioned in the question, then you’re probably not going in the right direction. In our situation, given that the axioms are consistent, then we know that the path to the solution will involve at least one resolution of the negated theorem statement with something else. Of course, the result of that resolution might also lead to the solution, and so on. This indicates the heuristic most used in set of support strategies: start with just the negated theorem in the SOS and keep adding in those clauses which are generated from the resolution of anything with an SOS member.

This strategy provides similar guide as backward chaining does

* + - * 1. Input resolution:  
           Every resolution combines one of the input sentences (from the KB or the query) with some other sentence. This strategy is only complete for knowledge base in Horn form.   
           Linear resolution (a complete strategy):  
           A generalization of input resolution, it relaxes the constraints on the one of the input sentences, allowing it to be an ancestor of the other sentence. E.g. P and Q to be resolved together either if P is in the original KB or if P is the ancestor of Q in the proof tree.
        2. Subsumption:  
           Eliminate all sentence that are subsumed by an existing sentence in the KB. One clause, C, is subsumed by another, D, if C is more specific than D.
        3. An relative paper: [Resolution Theorem Proving](https://www.doc.ic.ac.uk/~sgc/teaching/pre2012/v231/lecture9.html)

## Classical Planning

### Concept

#### Planning Domain Definition Language (PDDL)

A artificial intelligent planning language, which is used to deal with propositional logic

* + - * 1. State

A conjunction or set of fluents that are ground, functionless atoms. No variable is allowed, neither are function symbols and negations

[Database semantics](omnioutliner:///open?focus=lBjOXPdj9D2&row=heYiMWkH7At) is used to ensure missing fluents is regarded as false by default

* + - * 1. Action schema

Which consists of the action name, preconditions and effects

The preconditions and effects look like terms in first order logic at first glimpse since there are variables involved. That’s because they’re in a schema. When we apply the schema to ground states

E.g., An action schema for flying a plane from one location to another  
Action(Fly(p, from, to),  
 Precondition: At(p, from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)  
 Effect: ¬At(p, from) ∧ A(p, to))

* + - * 1. Result

The result of executing action a in state s is

Result(s, a) = (s - delete\_list(a)) ∪ add\_list(a)

The delete list of a is the negative literals in a’s effects

The add list of a is the positive literals in a’s effects

#### Decision problem

A problem that can be posed as a yes-no question of the input value

#### Turing machine

A theoretical machine that is used in thought experiments to examine the abilities and limitations of computers.

* + - * 1. Deterministic Turing machine

In which at most one action could be performed for a situation

* + - * 1. Non-deterministic Turing machine

In which more than one action could be performed for a situation

#### The complexity class

* + - * 1. PSPACE

The set of all decision problems that can be solved by a Turing machine (it turns out a non-deterministic Turing machine doesn’t add any extra power) using a polynomial amount of space.

* + - * 1. NP

The set of all decision problems that can be solved by a non-deterministic Turing machine in polynomial time

* + - * 1. P

The set of all decision problems that can be solved by a deterministic Turing machine in polynomial time

* + - * 1. What’s the relationship between PSPACE, NP and P?

PSPACE contains NP, which contains P

### Algorithm

#### Search

* + - * 1. Forward / Progression search

Which starts from the initial state and ends with the goal state.

Although this is prone to result in large state spaces, the performance can gain great improvement with the help of accurate heuristic

Enumerate general heuristics for planning

From the relaxed problem

Relax the constraints: ignore preconditions, ignore delete list

Although such heuristics help simplify the problems, but it is still expensive to calculate the heuristic function

From subproblems

In which the goal is reduced

Form a state abstraction: map many ground states to an abstract state—so as to cut down the search space

Such mapping procedures can be done by reducing the fluents

[Pattern database](omnioutliner:///open?focus=p2tmdggyJMs&row=pzUBIrM5mpp) provides a good example

Decomposition: divide a problem into parts

We assume subgoal independence: each part is independent from each other. This assumption can be both optimistic and pessimistic. It's optimistic when there are negated interaction between the subplans for each subgoal. It's pessimistic when several actions in subplans could be replaced by a single action in the merged plan.

When the assumption holds, the original problem can be solved by solving each subproblem independently and then combining the results.

When the assumption is not optimistic, the sum of costs of solving each subproblem provides a good heuristic

When the assumption is optimistic, the best we can do is to take the maximum of the costs as a heuristic. This heuristic, however, could be too low in most cases

* + - * 1. Backward / Regression search

Which starts from the goal state and ends with the initial state

What’s the result of regression from state g over action a?

s = (g - add\_list(a)) ∪ precondition(a)

* + - * 1. What results in the difference of progression search and regression search?

Regression search selects actions whose effects have intersection with the goals  
Progression search chooses actions whose preconditions meet the state

Regression search is more like dealing with satisfiability problem: We have the goal, and then we make up things based on that goal until some of those make-ups meets the initial state  
Progression search is more like dealing with entailment: From what we have at the start, we make inference until the goal is met

* + - * 1. What’s the advantage/disadvantage of backward search over forward search?

Backward search keeps the branching factor lower than forward search, for most problem domain. However, the fact that backward search uses state sets rather than individual states makes it harder to come up with good heuristics. That’s the main reason why the majority of current systems favor forward search

State sets result from the fact that, in order to infer a conclusion, backward search needs to recursively infer all the premises in some rule which contains that conclusion.

#### Classical planning

* + - * 1. Graph plan

Unfamiliar point. An algorithm which iteratively construct a planning graph, trying to extract a solution at each level by regression

Planning graph

A directed graph organized into levels. Levels include stage levels and action levels interleaved in the graph

What heuristics can we get from planning graph?

Max-level heuristic: return the maximum level cost of any of the goals

This is admissible but not necessarily accurate

Level sum heuristic: return the sum of the level costs of the goals

This is inadmissible but works well for problems largely decomposable

Set-level heuristic: return the level at which all the literal in the conjunctive goal appear in the planning graph without any pair of them being mutually exclusive

This is admissible and works extremely well on tasks in which there is a good deal of interaction among subplans

What can we obtain from planning graph?

If any goal fails to appear in the final level of the graph, the problem is unsolvable.

The level cost of a goal is an admissible and accurate heuristic for a goal.

The level cost of a goal is the level at which the goal first appears in the planning graph

Extract a plan directly from the planning graph

What does it mean for a literal g to appear at level S\_{i} in the planning graph?

If g appears at S\_{i}, there might exist a plan to achieve g at time i. If not, there is no such a plan

Serial planning graph

A planning graph adding mutexes between every pair of nonpersistent actions so that only one action can occur at a step.

Algorithm Overview

function graph-plan(problem) returns solution or none  
 initialize the planning graph with the initial state and an empty hash table for no-goods  
 while true   
 if all goals are not mutex with each other in current state level solution = extract a solution from the planning graph  
 if solution return solution  
 if graph and no-goods have both leveled off return none expand the planning graph

What does each stage level contain in planning graph? And for each action level?

Each stage level S\_{i} contains all literals that could hold at time i

Each action level A\_{i} contains all actions that could have their preconditions satisfied at time i

Describe the mutex for actions

Inconsistent effects: one action negates an effect of the other

Interference: one of the effects of one action is the negation of a precondition of the other

Such actions are prohibited because there is no strong guarantee that says which action will be taken first — when talking about that actions can be run simultaneously, we just mean the actions are partially ordered

Competing needs: one of the preconditions of one action is mutually exclusive with a precondition of the other

Two mutex precondition might be inconsistent support, so it’s not sufficient to just check whether one is the negation of the other

Describe the mutex for literals

One is the negation of the other

Inconsistent support: Each possible pair of actions that could achieve the two literals is mutually exclusive

What does no-goods contain?

(level, goals), where goals are a list of sets of goals not achievable at the level.

What’s the benefit of choosing (level, goals) as no-goods

It helps avoid redundant searchings: when the procedure of extracting a solution is called again with the same level and the same goals, it returns failure immediately

When to add a no-good to no-goods?

For the case that no-goods is (level, goals): also in extract, but this time it is done multi-times. As suggested below extracting a solution could be regarded as a backward search problem, every time it finishes checking a state and and fails to come up with a plan, it records the level and the set of goals it aims to achieve as a no-good

How to extract a solution from the planning graph?

Regard it as a Boolean constraint satisfaction problem, where the variables are the actions at each level, the values for each variable are either in or out of the plan, and the constraints are the mutexes and the need to satisfy each goal and preconditions

Regard it as a backward search problem, where each state in the search contains a pointer to a level in the planning graph and a set of goals it aims to achieve

How to choose the relevant action when do backward search?

Choose as candidates actions those of which at least one of the effects could unify with one element of the goal and none of effects negates an element of the goal

When does the planning graph level off?

The planning graph levels off when its current state level is the same as its previous state level

When does the no-goods level off?

For the case that no-goods is (level, mutexes): when two adjacent no-goods have the same size of goals

An implementation of Graph Plan

function extract(Graph, subGoals, level) return a plan, or failure  
 if level is the initial level return an empty plan  
 if subGoals is in no-goods return failure  
 call gpSearch to get a plan  
 if plan exists return plan  
 record subGoals as a no-good  
 return failure

A backward search to create a plan for achieving subGoals at the level. It records a no-good when there isn’t one

function gpSearch(Graph, subGoals, plan, level) return a plan, or failure  
 If subGoals is empty # a plan has generated by previous recursions  
 create a newGoals with preconditions of actions at level in plan  
 call extract(Graph, newGoals, level-1) to create a newPlan for newGoals in level-1  
 if newPlan is empty  
 return none  
 else   
 combine newPlan with plan and return  
 else   
 extract actions at level which are not mutex with actions in plan and whose effects include subGoals[0]   
 for action in actions create a newSubGoals which is the result of removing goals in subGoals achieved by action  
 create a planClone that is plan plus action at level  
 call gpSearch(Graph, newSubGoals, planClone, level) to crate a newPlan if newPlan exists return newPlan return failure

A recursive function to construct a plan. It recursively calls itself until subGoals is empty, and then call extract to create a plan for the previous level.

The plan passed in as argument is not the same as the plan returned: The first one just contains actions that can be executed, without mutually exclusive with each other, at the current level, while the later one contains actions from the initial level to the current level

What kind of problems do planning graphs work for?

Planning graphs only work for propositional problems—one with no variables

* + - * 1. [SAT plan](omnioutliner:///open?focus=fWEkKQsTkaU&row=hLZAjlA4AmC)
        2. Situation calculus

A logic formalism designed for representing and reasoning about dynamical domains

Action

Which consists of a action name and some variables as its argument

Situation

Which represents a history of action occurrence

The initial situation can be represented by s0, and the following situations should follow the recursive definition Result(s, a), where s is a situation and a is an action

Two situations are equal iff their start and actions are equal

Fluent

Property of world, which always takes a situation as their final argument

Relational fluent

Predicate that returns truth value

Functional fluent

Function that returns a situation-dependent value

Axioms

Possibility axiom

Which says when the action is possibly taken or under what precondition the action is possibly taken

It has the form

Preconditions(s) ⇒ Poss(a, s)

Preconditions(s) is some formula involving s that describes the preconditions and Poss(a, s) says action a is possibly taken at situation s

Example of a possibility axiom for an action fly(p, x, y), which says plan p flies from x to y:  
Plane(p, s) ∧ Airport(x, s) ∧ Airport(y, s) ∧ At(p, x, s) ⇒ Poss(Fly(p, x, y), s)

Successor-state axiom

Which says what happens to the fluent, depending on what action is taken

It has the form

Poss(a, s) ⇒ (Fluent is true ⇔ a’s effect made it true ∨ it was true before and action left it alone)

Example of a successor-state axiom for a fluent in(c, p, s), which says cargo c is in plane p in situation s  
Poss(a, s) ⇒ In(c, p, result(s, a)) ⇔ (a=Load(c, p) ∨ (In(c, p, s) ∧ a≠Unload(c, p)))

A solution is a situation (and hence a sequence of actions) that satisfies the goal

Why has situation calculus not been used pervasively?

There have not been any practical large-scale planning programs based on logical deduction over the situation calculus. This is in part because of the difficulty of doing efficient inference in FOL, but is mainly because the field has not yet developed effective heuristics for planning with situation calculus

* + - * 1. Encoding a planning problem as a constraint satisfaction problem
        2. Partial-order plan

It starts with an empty plan consisting of just the initial state and the goal, then it looks for a flaw in the plan and makes an addition to the plan to correct the flaw. A flaw is anything that keeps the partial plan from being a solution. At every step, we make the least commitment to fix the flaw, we don’t do anything more than those that need to be done. It backtracks when hitting a dead end.

### Q & A

#### What’s the major difference between search and planning?

* + - * 1. Algorithms used in search and planning are very similar. There is a significant difference, however: Planner deals with factory representations while searching algorithms deal with atomic representations

#### What’s the complexity of answering the question of whether there exists any plan that solves a planning problem?

* + - * 1. It’s in PSPACE, but if we disallow negative preconditions, it reduces to P

#### What’s the difference between partial-order plans and total-order plans?

* + - * 1. Partial-order plans leave decision about the ordering of actions as open as possible
        2. Total-order plans produce an exactly ordering of actions

## Planning and Acting in The Real World

### Terminology

#### Aggregation

The idea to group individual objects into quantities when the objects are all indistinguishable with respect to the purpose at hand

E.g.

Assume the actions in the problem take several people to do the job, in that case, we don’t name each people in the representation of resource, instead, we just specify how many people are available for the problem

#### Demonic nondeterministic

Nondeterministic caused by choice made by an adversary

#### Angelic nondeterministic

Nondeterministic caused by choice made by agent itself: e.g. for an HLA, different refinements may have different side effects

### Concept

#### Scheduling

Planning with temporal constraints

* + - * 1. Makespan

The total duration of a plan

* + - * 1. Critical path method

An algorithm for scheduling a set of project activities

Critical path

The path in partial-order plan whose total duration is longest

Slack of an action

For actions off the critical path, there’s a window of time in which they can be executed. Such a window is specified in terms of an earliest possible start time, ES, and a latest possible start time, LS. The quantity LS - ES is known as the slack of an action

By definition, actions on the critical path have no slack

What’s the formulae of ES, LS for actions in a partial-order plan?

The earliest possible start time of an action B is equal to the latest possible end time of actions taken before B

The latest possible end time of an action A is the reverse, which is equal to the earliest possible start time of actions taken after A

Thus, the latest possible start time of an action A, is its latest possible end time minus its duration

What happens to critical path problem when resource constraints are introduced?

Critical path problems are easy to solve because they have specified the orderings of some of actions (such orderings are introduced by the problem it self or by a planning procedure), and allow those unspecified actions to execute simultaneously. Things change when resource constraints are introduced: actions allowed to execute simultaneously before now compete for resources, and thus implicitly introducing some kind of undetermined ordering. That makes scheduling with resource constraints NP-hard

### Algorithm

#### Minimum slack

An algorithm dealing with scheduling with resource constraints

* + - * 1. On each iteration, schedule for the earliest possible start whichever unscheduled action has all its predecessors scheduled and has the least slack; then update the ES and LS times for each affected action and repeat

Although it’s not an optimal heuristic, it often works well in practice

#### Hierarchical task network (HTN)

An approach to automated planning in which the dependency among actions can be given in forms of networks

* + - * 1. Primitive action

Actions that are executable and can appear in the final plan

* + - * 1. High-level action (HLA)

Abstract actions, also regarded as goals, that require further decompositions to be executed

Refinement

A refinement contains a sequence of actions to achieve a high-level action, and in the book, it has form:

Refinement(High-level action,

Precondition(if there is one): …

Steps:[actions])

Actions in the Steps can be either primitive actions or high-level actions.

An HLA has one or more possible refinements. Different refinements for an HLA may have different Preconditions and different Steps, even though they have the same High-level action, their effects are not necessarily to be the same: they all achieve the same goal the HLA requested but meanwhile may have different side effects

Implementation

An implementation of high-level action is a refinement that contains only primitive actions

Downward refinement property

An ideal property saying that every high-level action which claims to achieve the goal does in fact achieve the goal

* + - * 1. How can we tell if a high-level plan achieves the goal from a given state

A high-level plan achieves the goal from a given state if at least one of its implementations achieves the goal from that state

The “at least one” is crucial — not all implementations need to achieve the goal, because the agent gets to decide which implementation it will execute

If its reachable set intersects the goal, the plan works. Otherwise, not.

* + - * 1. How to decide which implementation to take when HLAs have different implementations?

To search among the implementations for one that works

Why do we need to search, i.e. why would some implementation fail to work?

Although the HLA specifies the preconditions and effects, different refinements still have their own extra different preconditions and side effects. Such side effects would result in different states and such extra preconditions may not always be satisfied.

To reason directly about the HLAs — despite the multiplicity of implementations

This method resort to reachable set to enable the derivation of provably correct abstract plans, without the need to consider they implementations

* + - * 1. How does an agent construct a plan library containing known methods for implementing complex, high-level actions

One method is to learn the methods from problem-solving experience.   
One important aspect of this learning process is the ability to generalize the methods that are constructed, eliminating detail that is specific to the problem instance and keeping just the key elements of the plan.

Methods for achieving this kind of generalization are described in Chapter 19

* + - * 1. Hierarchical search

A BFS implementation

function hierarchical-search(problem) return a solution, or failure frontier = an empty FIFO queue frontier.push(the highest level plan)  
 while true if frontier.empty() return failure  
 plan = frontier.pop() prefix, hla, suffix = plan.prefix, plan.hla, plan.suffix  
 outcome = Result(problem.initial\_state, prefix) if hla is null then // plan is primitive   
 if outcome satisfies problem.goal return plan  
 else  
 for each refinement in Refinements(hla) sequence = concatenate(prefix, refinement.steps, suffix) front.push(Plan(sequence))

frontier: a FIFO queue containing plans

plan: a sequence of actions, it has three parts:

1. prefix: a subsequence of primitive actions before the first HLA in the sequence;

2. hla: the first HLA in the sequence;

3. suffix: The rest actions

if a plan contains only prefix, it’s primitive and is a leaf node in the search tree

outcome: the resulting state after applying prefix to the initial state of the problem

sequence: a list of actions used to construct plan

What’s the relationship between a node and its children?

A node is a plan, which contains a sequence of actions. Each child of a node is the consequence of applying decomposition to the first HLA in that node. For example, assuming a node contains a sequence of actions—[a[1], …, a[k-1], a[k], a[k+1], …], where a[k] is the first HLA in the sequence, each child of that node replaces a[k] with steps in a refinement and leaves other unchanged. So the number of children for a node is the number of refinements for the first HLA in the node

#### Angelic semantics for HLA description

Which specifies all possible side effects of an HLA

* + - * 1. Reachable set

Notation: Reach(state, hla). The set of states that can be reachable by some HLA’s implementations starting from state. This concept can be extended to a sequence of HLAs

Optimistic reachable set

Notation: Reach^+(state, hla). An approximate description that overstate the reachable set

Pessimistic reachable set

Notation: Reach^-(state, hla). An approximate description that understate reachable set

Whats the relationship among the reachable set, optimistic description and pessimistic description of an HLA?

Reach^-(state, hla) ⊆ Reach(state, hla) ⊆ Reach^+(state, hla)

How do we judge whether a high-level plan reaches the goal if we use approximate descriptions rather than exact one?

If the optimistic reachable set doesn’t intersect with the goal, the plan doesn’t work  
If the pessimistic reachable set interests with the goal, the plan works  
If the optimistic set intersects the goal but the pessimistic one doesn’t, the plan needs to be further refined

Why do we need the optimistic and pessimistic description while we already have the reachable set?

Because an HLA may have infinitely many implementations and may produce arbitrarily wiggly reachable sets. In that case, the best we can do is to estimate the reachable set so as to avoid being overwhelmed by details

* + - * 1. Angelic search

Because of the additional information, i.e. the optimistic/pessimistic reachable sets, angelic search searches shallower than hierarchical search. Otherwise, they’re almost identical

function angelic\_search(problem, highest\_level\_plan) return a primitive plan or failure frontier = an empty FIFO queue frontier.push(highest\_level\_plan)  
 while true  
 if frontier.empty() return failure plan = frontier.pop()  
 if reach^+(problem.initial\_state, plan) intersects problem.goal  
 if plan is primitive return plan guaranteed = reach^-(problem.initial\_state, plan) ∩ problem.goal // if the pessimistic reachable set intersects problem.goal  
 // the plan is guaranteed to reach the goal, decompose the plan  
 if guaranteed is not empty  
 finalState = any element of guaranteed return decompose(problem.initial\_state, plan, finalState)   
 prefix, hla, suffix = plan.prefix, plan.hla, plan.suffix  
 for each refinement in Refinements(hla) sequence = concatenate(prefix, refinement.steps, suffix) front.push(Plan(sequence))

Decompose a highest\_level\_plan to produce an implementation that satisfies problem.goal

This procedure may fail since the high\_level\_plan isn’t guaranteed to solve the problem

function decompose(initial\_state, plan, final\_state) return a primitive plan solution = an empty plan  
 while plan not empty action = plan.front() plan.remove\_front() s = a state in reach^-(initial\_state, action) such that final\_state ∈ reach^-(s, plan) problem = a problem with problem.initial\_state = initial\_state, problem.goal = s solution.append(angelic\_sarch(problem, action)) initial\_state = s

Decompose a plan to produce an implementation that starts at initial\_state and ends at final\_state

The plan is guaranteed to reach final\_state

* + - * 1. How to apply angelic search to least-cost problem?

Instead of a state being reachable or not, it has a cost for the most efficient way to get there. (The cost is ∞ for unreachable states.) The optimistic and pessimistic descriptions bound these costs. In this way, angelic search can find provably optimal abstract plans without considering their implementations

### Q & A

#### What does the classical planning representation deal with, and what’s it limit?

* + - * 1. Classical planning representation talks about what to do, and in what order, but it cannot talk about time: how long an action takes and when it occurs.

#### How to deal with time in planning?

* + - * 1. Plan first, schedule later: we divide the overall problem into a planning phase in which actions are selected, with some ordering constraints, to meet the goals of the problem, and a later scheduling phase, in which temporal information is added to the plan to ensure that it meets resource and deadline constraints

In the planning phase, we need a partial-order plan, not a total-order one. So the plans constructed by search based methods, such as graph plan should be converted to plans with minimal ordering constraint

#### What’s the difference between the classical planning and scheduling?

* + - * 1. Traditionally, the classical planning is done with specialized logical reasoning methods
        2. Traditionally, scheduling is done with constraint satisfaction, linear programming, or OR methods

### TODO

#### 11.3-11.4

## Knowledge Presentation

### Terminology

#### Upper ontology

The general framework of concepts, similar to basic classes in OOP.

#### Category

A general structure that represents a type of objects

#### Subcategory

If a category A is a subset of another category B, then A is a subcategory of B

#### Taxonomy / Taxonomic hierarchy

A organized hierarchy for a set of objects

#### Disjoint

Two or more categories are disjoint if they have no members in common

#### Exhaustive decomposition

If any object belonging to a category also belongs to one of its subcategories, then these subcategories constitute an exhaustive decomposition of that categories

#### Partition

A disjoint exhaustive decomposition

#### Part of

A subset relation between objects

#### Bunch of

A set of objects, which can be used to describe certain properties related to quantity

#### Stuff / Substance

A type of objects that’s uncountable

#### Things / Individual objects

A type of objects that’s countable

#### Intrinsic property

Properties that are retained under subdivision, such as density, color, ownership, etc.

#### Extrinsic property

Properties that are not retained under subdivision, such as weight, length, shape, etc.

#### Propositional attitude

Attitudes that an agent can have toward mental objects, such as Believes, Knows, Wants, Intends, and Informs

### Concept

#### Event calculus

A logical language for representing and reasoning about events and their effects

* + - * 1. Fluent

In the event calculus, fluents are reified. This means that they are objects and don’t have truth value.

* + - * 1. Event

Events are also terms. The effects of events are given using the predicates Initiates and Terminates

Discrete event

Events that have definite structure, such as duration, and cannot be divided, otherwise it becomes something different

Process / Liquid event

Events. If a process holds over an interval, it holds over any subinterval

* + - * 1. Predicates

T(f, t)

Assert whether fluent f is true at time t

Happens(e, i)

Assert whether event e happens over the time interval i

Initiates(e, f, t)

Assert whether event e causes fluent f to start to hold at time t

Terminates(e, f, t)

Assert whether event e causes fluent f to cease to hold at time t

Clipped(f, i)

Assert whether fluent f ceases to be true at some point during time interval i

Restored(f, i)

Assert whether fluent f becomes true sometime during time interval i

* + - * 1. Time intervals

Two kinds of time intervals

Moment

Time interval with zero duration

Extended Interval

Relations

Meet(i, j)

End(i) = Begin(j)

Before(i, j) / After(j, i)

End(i) < Begin(j)

During(i, j)

Begin(j) < Begin(i) < End(i) < End(j)

Overlap(i, j)

Begin(i) < Begin(j) < End(i) < End(j)

Begins(i, j)

Begin(i) = Begin(j)

Finishes(i, j)

End(i) = End(j)

Equals(i, j)

Begins(i, j) ∧ Finishes(i, j)

#### Modal logic

A type of logic that deals with possible worlds

* + - * 1. In a model-logic graph, nodes are possible worlds, two nodes are connected if everything in one world is consistent with the other, we call this accessibility relations. All worlds accessible for each other form a belief state

#### Monotonicity

In logic, monotonicity requires all entailed sentences to remain entailed after new sentences are added to the KB

#### Circumscription

An enhanced version of the closed-world assumption in which particular predicates (some unspecified information) are assumed to be false for every object except those for which they are known to be true, we say those predicate are to be circumscribed

* + - * 1. Example:  
           Bird(x) ∧ ¬Abnormal(x) ⇒ Flies(x)

here Abnormal is circumscribed

#### Model preference logic

In which a sentence is entailed if it’s true in all preferred models of the KB, as opposed to the requirement of truth in all models in classical logic

#### Default logic

A non-monotonic logic in which default rules can be written to generate contingent, non-monotonic conclusions

* + - * 1. It comprises of two parts.  
           1. The background theory, i.e., a set of facts  
           2. Default rules
        2. Default rules has form:  
           Prerequisite: Justification{1}, …, Justification{2} / Conclusion  
           According to this default, if we believe Prerequisite is true, and all Justification{i} is consistent with the background theory (i.e. not known false in the background theory), the Conclusion can be drawn

#### Belief revision & Belief update

* + - * 1. Both are the process of changing beliefs. The difference is:   
           belief revision occurs when new information is added to the KB, while belief update occurs when information in the KB is revised
        2. How to implement belief revision & belief update?

Systems designed to handle these problems are called truth maintenance systems (TMS)

The justification-based truth maintenance system (JTMS)

In which each sentence in the KB is annotated with a justification consisting of the set of sentences from which it was inferred. If the information added or revised violates a member of a justification, the justification is removed. And if the justification is the only one for a sentence, the sentence is also removed.

Example:  
Assume sentence Q has the single justification {P, P ⇒ Q}, later P is revised to be false, which causes the justification {P, P ⇒ Q} is removed, which in turns causes Q is removed from the KB

For the case of belief update, JTMS generally don’t remove a sentence from the KB when it loses all justifications, instead, it marks sentences as out of the KB so that when a subsequent assertion restores one of the justification, the sentence can be simply back in without any inference

Assumption-based truth maintenance system (ATMS)

It keeps check, for each sentence, of a set of assumption sets that would cause the sentence to be true. A sentence holds when one of its assumption sets are satisfied

#### Belief update

The process of changing beliefs when a knowledge base is revised to reflect a change in the world

### Q & A

#### How to represent categories in first-order logic?

* + - * 1. Two ways:  
           1. Predicates: Basketball(b)  
           2. Objects: Basketballs

#### How to express, in first-order logic, the fact

* + - * 1. An object (BB9) is a member of a category (Basketballs)?

BB9 ∈ Basketballs

* + - * 1. A category (Basketballs) is a subclass of another category (Balls)?

Basketballs ⊂ Balls

* + - * 1. All member of a category (Basketballs) have some properties (Spherical)?

(x ∈ Basketballs) ⇒ Spherical(x)

* + - * 1. Members of a category (Basketballs) can be recognized by some properties (Orange, Round, Diameter, Balls)?

Orange(x) ∧ Round(x) ∧ Diameter(x)=9.5'' ∧ x ∈ Balls ⇒ x ∈ Basketballs

* + - * 1. A category (Dogs) as a whole has some properties (DomesticatedSpecies)

Dogs ∈ DomesticatedSpecies

DomesticatedSpecies is a category of categories



#### How to express Disjoint(s), ExhaustiveDecomposition(s, c), Partition(s, c) in first-order logic?

s is a set of categories, c is a category

* + - * 1. Disjoint(s) ⇔ (∀c1∈s, c2∈s c1 ≠ c2 ⇒ Intersection(c1, c2)={})
        2. ExhaustiveDecomposition(s, c) ⇔ (∀i∈c ∃c2∈s i∈c2)
        3. Partition(s, c) ⇔ Disjoint(s) ∧ ExhaustiveDecomposition(s, c)

#### What’s the difference between PartOf and ∈?

* + - * 1. Both arguments of PartOf are objects, while one of arguments of ∈ is a category

#### What’s the goodness in qualitative measures?

* + - * 1. Although qualitative measures cannot be assigned a numerical value as quantitative measures, they can be ordered

#### What kind of properties stuff and things contains, respectively?

* + - * 1. Stuff only defines intrinsic properties, while things define both intrinsic and extrinsic properties

## Quantifying Uncertainty

### Terminology

#### Sample space

The set of all possible worlds, which are mutually exclusive and exhaustive. We use 𝛺 to refer to the sample space, and 𝜔 to refer to elements of the space

#### Event

A set of possible worlds that satisfy a certain proposition

#### Unconditional probability / Prior probability / Prior

Degree of belief in proposition in the absence of any other information

#### Conditional probability / Posterior probability / Posterior

Degree of belief in proposition given some evidence

#### Random variable

A variable whose possible values are numerical outcomes of a random phenomenon.

When we discuss random variables, we always use words beginning with uppercase to denote random variable, and words all in lowercase to denote its values

### Concept

#### Qualification problem

The problem concerned with the impossibility of listing all the preconditions required for a real-world action to have its intended effect.

#### Decision theory

The combination of probability theory and utility theory

#### Maximum expected utility (MEU)

An agent is rational iff it choose the action that yields the highest expected utility, averaged over all the possible outcomes of the action

#### Probability distribution

The probability distribution for a random variable (𝑷(variable)) is the probabilities for values in the random variable’s domain. It’s a vector if we predefine the ordering of the values

#### Probability density function (PDF)

A function, whose value at any given sample in the sample space can be interpreted as providing a relative likelihood that the value of the random variable would equal that sample

* + - * 1. What’s the difference between probabilities and probability density functions?

Probabilities are applied to variables with finite domain, while PDFs are to variables with infinite domain

Probabilities are unitless number, whereas PDF are measured with a unit

For example, assuming a car’s speed is uniformly distributed between 30km/h and 60km/h, the probability density function of speed = 40km/h is 1/30 h/km, but the probability of that is 0 because it is a region of width 0

#### Joint probability distribution

Given at least two random variable X, Y, …, that are defined on a probability space, the joint probability distribution for X, Y, ..., 𝑷(X, Y, …), is a probability distribution that gives the probability that each of X, Y, ... falls in any particular range or discrete set of values specified for that variable

* + - * 1. Full joint probability distribution

The joint probability distribution for all the random variables

#### Marginal probability

The marginal probability of a random variable or some set of variables is the distribution over it.

* + - * 1. Marginalization / Summing up

P(Y)=∑\_{z∈Z}P(Y, z), Z is the set of random variables other than Y

* + - * 1. Conditioning

P(Y)=∑\_{z∈Z}P(Y|z)P(z), Z is the set of random variables other than Y

#### Normalization

The process converting 𝑷(A, B) to 𝑷(A|B)

* + - * 1. Normalization constant / Normalizer

𝑷(A|B) = 𝑷(A, B) / 𝑷(B), 1/𝑷(B) is the normalization constant for the distribution 𝑷(A|B), which ensures that it adds up to 1

#### Cause & Effect

* + - * 1. Causal probability

P(effect|cause)

* + - * 1. Diagnostic probability

P(cause|effect)

* + - * 1. Why do we calculate the diagnostic probability from causal probability?

Cause is usually unobservable, so we generally don’t know which case of causal probability to calculate, and the best we can do is to draw such a probability from samples. Effect, on the other hand, is observable, so we know exactly which case of diagnostic probability to calculate. Furthermore, extracting diagnostic probabilities from statistical information is fragile, while extracting causal probabilities is pretty stable, as explained [here](omnioutliner:///open?focus=fqbU-o8bU3s&row=mk_tW19GeFN)

For example, assuming cause is meningitis and effect is some test, we don’t know whether a patient has meningitis, so we don’t know whether the current situation corresponds to P(test|meningitis) or P(test|¬meningitis), but we can draw both from experience.

* + - * 1. How to derive diagnostic probability from causal probability?

𝑷(Cause|Effect) = 𝑷(Effect|Cause) 𝑷(Cause) / 𝑷(Effect)

* + - * 1. Why do we use Bayes' rule to infer diagnostic probability instead of extracting it from statistical information?

Extracting diagnostic probabilities from statistical information is fragile. Unlike causal probabilities, which is pretty stable. Diagnostic probabilities should go up and down as P(cause) goes up and down (since P(cause|effect) = P(effect|cause)P(cause) / P(effect)), but such a feature may not immediately embody in statistical information.

For example:

If an epidemic of meningitis suddenly happens, P(meningitis) will go up and the diagnostic probability, P(meningitis|test), is also supposed to go up. A doctor who gains the diagnostic probability P(meningitis|test) from statistical observation before the epidemic will have no idea how to update P(meningitis|test), but a doctor who computes P(meningitis|test) using Bayes’ rule will see that P(meningitis|test) goes up proportionately with P(meningitis)

#### Naive Bayes classifier

Which assumes all the effects are independent of each other conditioned on some cause, i.e. 𝑷(Cause, Effect{1}, Effect{2}, …) = 𝑷(Cause) ∏\_{i}𝑷(Effect{i}|Cause)

In practice, naive Bayes system can work surprisingly well, even if the conditionally independence assumption is not true

### Q & A

#### What’s the drawbacks of agents with a belief state?

* + - * 1. The belief state could be impossibly large and complex
        2. The contingent plan therefore grows arbitrarily large
        3. There may be not a plan which could cope with every possible world, i.e. no plan guaranteed to achieve the goal

#### Why not use logic to cope with uncertainty?

* + - * 1. Laziness: It is too much work, almost impossible to list the complete set of antecedents or consequents needed to ensure an exceptionalness rule and too hard to use such rules
        2. Theoretical ignorance: We don’t have the complete theory for the domain
        3. Practical ignorance: Even if we have the complete theory, we might still be uncertain about a particular individual because some information has not been acquired or can be unobtainable

#### In the real world, a thing either happens or doesn’t, what’s the point of probability?

* + - * 1. Probability provides a way of summarizing the uncertainty that comes from our laziness and ignorance. Probability statements are made with respect to a knowledge state, not with respect to the real world. That is, probability describes the likelihood that things will happen based on the knowledge (history, or something else)

#### What’s the rational decision in a uncertain environment?

* + - * 1. The rational decision depends on both the relative importance of various goals and the likelihood that, and degree to which, they will be achieved. ([MEU](omnioutliner:///open?focus=fqbU-o8bU3s&row=hJvAd64Vmsi))

#### How does an agent make tradeoff between different choice?

* + - * 1. An Agent has preferences between the different possible outcomes of various plans. Utility is used, by an agent, to represent and reason with preferences

#### Why don't we use full joint probability distribution as a tool for building realistic reasoning systems?

* + - * 1. It grows exponentially large in the number of random variables, so it’s impractical in realistic situations

#### When are two events independent?

* + - * 1. When they share no common cause, or no common ancestor in a Bayesian network.

#### What kind of condition could make two dependent events conditional independent?

* + - * 1. When the condition covers the common cause, i.e. the common ancestor in the Bayesian network.
        2. Example  
           Assume A causes B and C, B causes D, and C causes E.  
           In this case, D and E are dependent because they share the common cause A. To make them conditionally independent, the condition could be either A, B, or C.

#### When is the conditional independence useful?

* + - * 1. It is troublesome to calculate 𝑷(Effect{1}, Effect{2}…), which is 𝑷(Effect{1}|Effect{2}, …) 𝑷(Effect{2}|Effect{3}, …) …, when all the Effects are dependent, but if we can find a common cause for all of these Effects, it’ll be pretty easy to calculate 𝑷(Cause, Effect{1}, Effect{2}, …), which is 𝑷(Cause) ∏\_{i}𝑷(Effect{I}|Cause). Now we can calculate 𝑷(Effect{1}, Effect{2}…) through marginalization

#### How to introduce dependency to two independent events?

* + - * 1. Condition them on a common successor

## Probabilistic Reasoning

### Concept

#### Bayesian network

A directed acyclic graph where

1. Each node corresponds to a random variable.

2. Two nodes are linked if one is the parent of the other, in other words, one is conditionally dependent on the other

3. Each node X has a conditional probability distribution 𝑷(X|Parents(X)) that quantifies the effect of all the parents on the node (that’s in cause direction, which is easy to be obtained from statistical information)

* + - * 1. Variable in Bayesian network

Query variable

The variable we query

Evidence variable

The variable whose value is assigned

Hidden variable

Non-query, non-evidence variable

* + - * 1. Two ways to consider Bayesian networks

View a Bayesian network as a representation of the joint probability distribution

For random variables X{1}, X{2}, …, ordered such that effects precedes causes, we repeatedly apply the product rule, then we have  
𝑷(X{1}, X{2}, …) = 𝑷(X{1}|X{2}…) 𝑷(X{2}|X{3}…)…  
This identity is called the chain rule. Furthermore, if we assume that Parents(X{i}) ⊆ {X{i+1}, …} are the direct causes of X{i}, then the above identity can be reduced to   
𝑷(X{1}, X{2}, …) = 𝑷(X{1}|Parents(X{1})) P(X{2}|Parents(X{2})) …  
Because elements in {X{i+1}, …} other than Parents(X{i}) are independent of X{i} conditioned on Parents(X{i})

How is this way helpful in constructing networks?

This way illustrates a parental relationship between nodes. That is, one is the parent of the other iff it’s the direct cause of the other. There is another advantage to construct networks in this way: causal probabilities are more robust than diagnostic probabilities, as discussed [here](omnioutliner:///open?focus=fqbU-o8bU3s&row=cuHU1seP0qX)

The parental relationship can also be based simply on whether two nodes are related. In that way, however, we may end up having too many links

It helps simplify the conditional probability table: Assuming a network of n random variables, each random variable is directed influenced by k others, then the amount of information needed to specify each conditional probability table will be at most 2^k numbers, and the complete network can be specified by n(2^k) numbers. In contrast, the joint distribution contains 2^n numbers

For small k, it’s a huge memory saving to use Bayesian network

View a Bayesian network as an encoding of a collection of conditional independence statements

Markov blanket

The Markov blanket for a node in a Bayesian network is a set of nodes composed of its parent, children, and children’s parent

Why does Markov blanket contain children’s parent?

Even in the case where a child’s parents are independent of each other, they are conditionally dependent given the child because one parent explains away another conditioned on their common child, as explained [here](omnioutliner:///open?focus=m-j2eC1Kgrl&row=lSK7jZN8l63)

For example, assume a child C, with parents A and S, is the minimum value of its parents, given C=0 and A=1, it’s easy to infer that S=0. As it shows, given C, the knowledge of A provides information of value of S. Thus, A is conditionally dependent on S given C

A node is conditionally independent of all its non-descendants given its parents

A node is conditionally independent of all other nodes in the network, given its Markov blanket

How does this help design inference procedure?

It helps to do approximate inference — since sampling a variable requires the probability of that variable conditioned on all other sampled variables, the conditional independence property of Bayesian network helps to simplify the calculation of such a probability

* + - * 1. Singly connected network / polytree

A Bayesian network whose underlying undirected graph is a tree (connected and acyclic)

What’s the time and space complexity of exact inference in polytrees?

It’s linear in the size of the network. Here, the size is defined as the number of CPT entries; if the number of parents of each node is bounded by a constant, then the complexity will also be linear in the number of nodes

It’s easy to find the optimal ordering for a polytree by rooting the tree at the query node and iterating through the nodes in post-order, that is eliminating children before its parent.

Note that the tree constructed here is merely a guide for ordering of variable elimination, factors are still constructed based on the original Bayesian network

* + - * 1. Multiply connected networks

A Bayesian network whose underlying undirected graph is cyclic

What’s the time and space complexity of exact inference in polytrees?

It’s exponential in the size of the network, even when the number of parents per node is bounded

* + - * 1. Why do we use Bayesian network?

Bayesian network helps reduce the number of probabilities we need to calculate the joint probability: the joint distribution over n variables requires 2^n - 1 probability values, but with help of Bayesian network, we can significantly reduce the number of values required, as discussed [here](omnioutliner:///open?focus=m-j2eC1Kgrl&row=cAaUfrkeLUt).

#### Bipartite graph

A graph whose vertices can be divided into two disjoint sets such that every edge connects a vertex in the first set to one in the second.

* + - * 1. Independent set | Stable set

A set of vertices in a graph, no two of which are adjacent.

#### Local structured system / Sparse system

A system in which each subcomponent interacts directly with only a bound number of other components, regardless of the total number of components

#### Probability distribution

* + - * 1. Gaussian distribution / Normal distribution

, where 𝜇 is the mean, 𝜎 is the standard deviation

Why is Gaussian distribution useful?

The normal distribution is useful because of the [central limit theorem](https://en.wikipedia.org/wiki/Central_limit_theorem). In its most general form, under some conditions (which include finite [variance](https://en.wikipedia.org/wiki/Variance)), it states that averages of samples of observations of [random variables](https://en.wikipedia.org/wiki/Random_variables) independently drawn from independent distributions [converge in distribution](https://en.wikipedia.org/wiki/Convergence_in_distribution) to the normal, that is, become normally distributed when the number of observations is sufficiently large. Physical quantities that are expected to be the sum of many independent processes (such as [measurement errors](https://en.wikipedia.org/wiki/Measurement_error)) often have distributions that are nearly normal. Moreover, many results and methods (such as [propagation of uncertainty](https://en.wikipedia.org/wiki/Propagation_of_uncertainty) and [least squares](https://en.wikipedia.org/wiki/Least_squares) parameter fitting) can be derived analytically in explicit form when the relevant variables are normally distributed.

Standard normal distribution

Linear Gaussian distribution

A Gaussian distribution whose 𝜇 varies linearly with another value and whose 𝜎 is fixed

Multivariate Gaussian

A Gaussian distribution over a vector x in n dimensions:

, where 𝜇 is the mean vector , 𝛴 is the covariance matrix

Covariance

The covariance of two random variable is the expectation of the product of their difference from their means

cov(X, Y) = E((X-𝜇\_{X})(Y-𝜇\_{Y}))

Covariance matrix

𝛴, a matrix of covariances between elements of a vector of random variables

Given a vector X = ⟨X{1}, …, X{n}⟩^⟙, the entries of the covariance matrix are as follows:

𝛴(i, j)=cov(X{i}, X{j}) = E((X{i} - 𝜇{i})(X{j} - 𝜇{j}))

#### Hybrid Bayesian network

A bayesian network with both discrete and continuous variables

* + - * 1. For a node which corresponds to a continuous variable or of which some parents correspond to continuous variables, we specify probability distributions rather than probabilities.

Considering a continuous variable, some of whose parents are continuous variables and the others are discrete variables, we specify one probability distribution (often a probability density function) for each assignment to the discrete variables. Such probability distribution cover how the distribution over the value of current variable depends on the value of its continuous parents

For a simplest example, assuming a continuous variable Cost, which has two parent — one is a continuous variable Harvest, the other is a Boolean variable Subsidy — we’re going to define two probability distributions — one for each assignment to Subsidy. The most common choice of probability distribution is linear Gaussian distribution (as explained in [Why is Gaussian distribution useful?](omnioutliner:///open?focus=m-j2eC1Kgrl&row=fdOU_ElNbDt)), in which the continuous variable Cost has a Gaussian distribution whose mean varies linearly with its parent Harvest and whose standard deviation is fixed. Thus, we have   
 Those parameters can be learned from sample space

Considering a discrete variable, some of whose parents are continuous variables and the others are discrete variables, we, as before, specify one probability distribution (often a probability density function) for each assignment to the discrete value. Such probability distributions state the same meaning as described in continuous variables

For examples, see Page 521-522.

What’s the most common choice of probability distribution for continuous variables in Bayesian network?

Define standard families of probability density functions that are specified by a finite number of parameters, such as Gaussian distribution

If a Bayesian network contains only variables with linear Gaussian distributions, what’s the joint distribution? And the posterior distribution?

Both are multivariate Gaussian distributions over all the variables

When discrete variables are added to a Bayesian network containing only variables with linear Gaussian distributions, what’s the conditional distribution for a continuous variable with discrete parents?

It’s a conditional Gaussian distribution. That is, given any assignment to the discrete variables, the distribution over the continuous variables is a multivariate Gaussian

#### Markov chain Monte Carlo

A class of algorithm for sampling from a probability distribution based on constructing a Markov chain that has the desired distribution as its [equilibrium distribution](https://en.wikipedia.org/wiki/Markov_chain#Steady-state_analysis_and_limiting_distributions). The state of the chain after a number of steps is then used as a sample of the desired distribution.

* + - * 1. Markov chain / Markov process

A stochastic process satisfying Markov property. That is, one can make predictions for the future of the process based solely on its present states, not on the history. Mathematically, 𝑷(S{t+1}|S{t}, …, S{0}) = 𝑷(S{t+1}|S{t}), where S{t} is the state at time t

* + - * 1. Markov property

The memoryless property of a stochastic process

* + - * 1. Transition probability

The transition probability P(s’|s) is the probability that the process make a transition from state s to state s’

* + - * 1. Stationary distribution / Equilibrium distribution

Let P\_{t}(s) be the probability of being in state s at time t, given the transition probability P(s|s), we calculate the probability of being in state s’ at time t+1 as below

P\_{t+1}(s’)=∑\_{s} P\_{t}(s) P(s’|s)

We say that the chain has reached its stationary distribution P, if P\_{t}{s}=P\_{t+1}{s}, for all state s. Its defining equation is

P(s’) = ∑\_{s} P(s)P(s’|s), for all s’

Which can be read as saying that the expected "outflow" from each state (P(s’)) is equal to the expected "inflow" from all the states (∑\_{s} P{s)P(s’|s))

What’s the connection between stationary distribution and [eigenvectors](#Row__)?

The stationary distribution can be written in a form as below  
 Where 𝑷(X) is an (n × 1) vector, P(X|X) is an (n × n) matrix, A, where A[i, j] is P(j|i),  
Thus, 1 is an eigenvalue of P(X|X), and 𝑷(X) is an eigenvector corresponding to 1

From the above observation, we can see that as long as |P(X|X) - I| = 0 — which is naturally held by Markov models because all numbers in P(X|X) is in the range of [0, 1] and the sum of each row is 1 — there is always a stationary distribution

Applications of stationary distribution

PageRank

Which ranks the pages so that a page with more income links has higher rank.

Initial distribution: uniform over all pages

1/n for each page in total n pages

Transition probability:  
With probability c, uniformly jump to a random page  
With probability 1-c, uniformly follow a random out-link

Go through this model, when it converges to its stationary distribution, the highly reachable pages will have high ranks

* + - * 1. Detailed balance

A transition probability P(s, s’) is in detailed balance with P(s) when

P(s) P(s’|s) = P(s’) P(s|s’), for all s, s’

Which could be viewed as the flow from s to s’ is equal to the flow from s’ to s, for all s and s’

* + - * 1. What’s the relationship between detailed balance and stationary distribution?

Detailed balance implies stationary distribution:  
Intuitively, if the flow from s to s’ is equal to the flow from s’ to s, for some s and all its neighbor s’, then the outflow from s is equal to its inflow  
Mathematically:  
∑\_{s} P(s) P(s’|s) = ∑\_{s} P(s’)P(s’|s) = P(s’) ∑\_{s} P(s|s’) = P(s’)

#### Relational probability model (RPM)

A model, which borrows idea from database semantics of first-order logic, in which unique name assumption and domain closure are made, but closed-world assumption is not

* + - * 1. Type signature

A commitment made by RPM for each function—a specification of the type of each each argument and the function’s value

* + - * 1. Dependency function

The conditional probabilities for function’s values

* + - * 1. Context-specific independence

Which allows a variable to be independent of some of its parents given certain values of others

* + - * 1. Relational uncertainty

Uncertainty in the function’s value, which affects the dependency structure of the network

For instance, in the following example, if Honest(C) is unknown, then the value of the conditional test in the Recommendation is unknown too.

Such uncertainty can be diminished by considering other relative information and doing some inference

* + - * 1. Examples:  
           Type signature:  
            Quality: Book -> {1, 2, 3, 4, 5}  
            Honest: Customer -> {true, false}  
            Kindness: Customer -> {1, 2, 3, 4, 5}  
            Recommendation: Customer, Book -> {1, 2, 3, 4, 5}  
           Dependency function  
            Quality(B) ~ <0.05, 0.2, 0.4, 0.2, 0.15>  
            Honest(C) ~ <0.99, 1>  
            Kindness(C) ~ <0.1, 0.1, 0.2, 0.3, 0.3>  
            Recommendation ~ CPT(Honest(C), Kindness(C), Quality(B))  
           Context-specific independence  
            Recommendation ~ if Honest(C) then  
            HonestCPT(Kindness(C), Quality(B))  
            else  
            <0.4, 0.1, 0, 0.1, 0.4>

#### Sibyl

Sibyls are multiple IDs referring to the same underlying object

#### Sibyl attack

The use of sibyls to confound a reputation system

#### Existence uncertainty

What are the real objects underlying the observed data

#### Identity uncertainty

Which symbols really refer to the same objects

### Algorithm

#### Exact inference

* + - * 1. Enumeration Ask

A recursive implementation of calculating 𝑷(X|𝐞), evaluating the expression from parents to children

The space complexity: O(n), since the only CPTs

The time complexity: O(2^n), where n is the number of Boolean variables

def enumeration\_ask(X, e, bn):  
 """Return the conditional probability distribution of variable X  
 given evidence e, from BayesNet bn. [Figure 14.9]  
 >>> enumeration\_ask('Burglary', dict(JohnCalls=T, MaryCalls=T), burglary  
 ... ).show\_approx()  
 'False: 0.716, True: 0.284'"""  
 """  
 Parameters:  
 X string: a random variable name  
 e dictionary: an event of form [random variable: value]  
 bn BayesNet: a Bayesian network containing only Boolean-value nodes  
 """  
 assert X not in e, "Query variable must be distinct from evidence"  
 Q = ProbDist(X)  
 for xi in bn.variable\_values(X):  
 # bn.variables are ordered with parents before children  
 Q[xi] = enumerate\_all(bn.variables, extend(e, X, xi), bn)  
 return Q.normalize()  
  
def enumerate\_all(variables, e, bn):  
 """Return the sum of those entries in P(variables | e{others})  
 consistent with e, where P is the joint distribution represented  
 by bn, and e{others} means e restricted to bn's other variables  
 (the ones other than variables). Parents must precede children in variables."""  
 if not variables:  
 return 1.0  
 Y, rest = variables[0], variables[1:]  
 Ynode = bn.variable\_node(Y)  
 if Y in e:  
 return Ynode.p(e[Y], e) \* enumerate\_all(rest, e, bn)  
 else:  
 return sum(Ynode.p(y, e) \* enumerate\_all(rest, extend(e, Y, y), bn)  
 for y in bn.variable\_values(Y))

For a concrete example, 𝑷(B|j, m) = α𝑷(B) ∑\_{e}P(e) ∑\_{a}P(a|B, e) P(j|a) P(m|a)

Above procedure calculates it from left to right — the variable ordering might be different since it hasn’t been well organized and solely depends on variables. The else part in enumerate\_all deal with the case where ∑ is involved

It is essential to realize that this algorithm enumerates variables from parents to children. Such an order helps to ensure Ynode.p is always calculable, since e has already specified the value of Y’s parents, and to avoid some redundant calculations caused by bad variable ordering: For instance, if we move P(B) inside ∑\_{e} in above example, P(B) will be unnecessarily calculated twice.

This procedure causes some redundant calculations resulting from the fact that an inner variable may not have connection with the outer variables. In our example, P(j|a) P(m|a) and P(j|¬a) P(m|¬a) have no connection with e, but they are calculated twice, one for e, the other for ¬e  
Such wastes can be avoided by dynamic programming: calculate the expression from right to left, and save the results for later use

* + - * 1. Variable Elimination

A dynamic-programming implementation of calculating 𝑷(X|𝐞), evaluating the expression from right to left so that it could save the results for later use

The time complexity is O(nd^s), where n is the number of non-evidence variables, d is the domain size of variables, and s is the maximum size of any factor during the process. Here, the size of a factor is defined as the number of non-evidence variables it contains, so d^s is the size of the factor’s conditional probability table under the assumption that all variables has the same domain size

Algorithm Overview

def elimination\_ask(X, e, bn):  
 """Compute bn's P(X|e) by variable elimination. [Figure 14.11]  
 >>> elimination\_ask('Burglary', dict(JohnCalls=T, MaryCalls=T), burglary  
 ... ).show\_approx()  
 'False: 0.716, True: 0.284'"""  
 assert X not in e, "Query variable must be distinct from evidence"  
 factors = []  
 for var in reversed(bn.variables):  
 factors.append(make\_factor(var, e, bn))  
 if is\_hidden(var, X, e):  
 factors = sum\_out(var, factors, bn)  
 return pointwise\_product(factors, bn).normalize()  
  
def is\_hidden(var, X, e):  
 """Is var a hidden variable when querying P(X|e)?"""  
 return var != X and var not in e

This algorithm consists of three steps:

1. Choose a query or hidden variable X, multiply all factors containing X to obtain factor F(X, …)

2. Eliminate X from F(X, …) to obtain factor F(…) by summing over its values

3. Replace all the factors containing X with F(…)

Why does this implementation choose to make factors from children to parents?

The algorithm eliminates variables while making factors, so it needs to make sure all factors whose variables includes the variable to be eliminated must has been made. Because a factor’s variables includes the variable passed in as an argument and its parents, but not its children, making factors from children to parents manages to satisfy the requirement.

Factor

A multi-dimensional table for 𝑷(\*) or P(\*), assigning a value to each corresponding variable. It contains 2^n entires where n is the number of query and hidden variables

Data member:

variables string: a list of random variables

cpt dictionary: the conditional probability table for variables of form [values of variables: conditional probability]

Member function:

pointwise\_product(other, bn): do self × other

sum\_out(var, bn): do ∑\_{var}self

p(e): return the conditional probability of the values of self.variables given in event e

class Factor:  
 """A factor in a joint distribution."""  
 def \_\_init\_\_(self, variables, cpt):  
 """  
 **:param** variables: variable name  
 **:param** cpt: dict(value\_of\_variable: probability)  
 """  
 self.variables = variables  
 self.cpt = cpt  
 def pointwise\_product(self, other, bn):  
 """Multiply two factors, combining their variables."""  
 variables = list(set(self.variables) | set(other.variables))  
 cpt = {event\_values(e, variables): self.p(e) \* other.p(e)  
 for e in all\_events(variables, bn, {})}  
 return Factor(variables, cpt)  
 def sum\_out(self, var, bn):  
 """Make a factor eliminating var by summing over its values."""  
 variables = [X for X in self.variables if X != var]  
 cpt = {event\_values(e, variables): sum(self.p(extend(e, var, val))  
 for val in bn.variable\_values(var))  
 for e in all\_events(variables, bn, {})}  
 return Factor(variables, cpt)  
 def normalize(self):  
 """Return my probabilities; must be down to one variable."""  
 assert len(self.variables) == 1  
 return ProbDist(self.variables[0],  
 {k: v for ((k,), v) in self.cpt.items()})  
 def p(self, e):  
 """Look up my value tabulated for e.  
 **:param** e: dict(variable: value), which specifies values for all variables  
 """  
 # event\_values here succeeds in separating e from Factor.variables  
 # so that we don't need to care about which variables Factor contains  
 # when inquiring probability -- we just need to pass in an event which  
 # includes self.variables  
 return self.cpt[event\_values(e, self.variables)]

def event\_values(event, variables):  
 """Return a tuple of the values of variables in event.  
 >>> event\_values ({'A': 10, 'B': 9, 'C': 8}, ['C', 'A'])  
 (8, 10)  
 >>> event\_values ((1, 2), ['C', 'A'])  
 (1, 2)  
 """  
 if isinstance(event, tuple) and len(event) == len(variables):  
 return event  
 else:  
 return tuple([event[var] for var in variables])

The essential part of the data structure, which extract the values of variables in event. With help of this function, we are able to inquire the probabilities of different factors (Factor.p(e)) using an event which specifies values of all variables. Although it simplifies tasks, it also introduces repeated calculations in construction of cpt, since many events generated by all\_events will be mapped into the same tuple of values using event\_values

def all\_events(variables, bn, e):  
 """Yield every way of extending e with values for all variables."""  
 if not variables:  
 yield e  
 else:  
 X, rest = variables[0], variables[1:]  
 for e1 in all\_events(rest, bn, e):  
 for x in bn.variable\_values(X):  
 yield extend(e1, X, x)

Make factor

Make a factor whose variables includes var and its parents which are hidden variables

def make\_factor(var, e, bn):  
 """Return the factor for var in bn's joint distribution given e.  
 That is, bn's full joint distribution, projected to accord with e,  
 is the pointwise product of these factors for bn's variables."""  
 node = bn.variable\_node(var)  
 variables = [X for X in [var] + node.parents if X not in e]  
 cpt = {event\_values(e1, variables): node.p(e1[var], e1)  
 for e1 in all\_events(variables, bn, e)}  
 return Factor(variables, cpt)

Pointwise product

Multiply all factors, and then return a factor whose variables is the combination of variables in all factors

The operations among factors must only be pointwise product

For the case where factors contain n distinct non-evidence variables and each variable has d values. This process takes O(d^n) time

def pointwise\_product(factors, bn):  
 return reduce(lambda f, g: f.pointwise\_product(g, bn), factors)

Sum out

Eliminate var from all factors by summing over its values

The operation among factors could only include pointwise product, no nested summation is considered

For the case where factors contain n distinct non-evidence variables and each variable has d values. The sum operation only takes O(d) time; if the pointwise product is involved, the whole process takes O(d\*d^n)

def sum\_out(var, factors, bn):  
 """Eliminate var from all factors by summing over its values."""  
 result, var\_factors = [], []  
 for f in factors:  
 (var\_factors if var in f.variables else result).append(f)  
 result.append(pointwise\_product(var\_factors, bn).sum\_out(var, bn))  
 return result

TODO: How to choose the ordering of elimination of hidden variables?

In fact every ordering of elimination of variables is valid, but different orderings yield different performance. To find the optimal one is NP-hard. There are, however, some heuristics we can resort to. All of these are trying to minimize the size of the factor constructed during the process

To choose some order other than the one illustrated in the above implementation, we may need to make all factors in advance

Min-neighbors: Choose a variable with the fewest dependent variables

Minimize the size of the resulting factor’s variable

Min-weight: Choose a variable to minimize the product of the cardinalities of its dependent variables

Minimize the size of the resulting factor’s conditional probability table

Min-fill: Choose vertices to minimize the size of the factor that will be added to the graph.

How to further improve the efficiency of the variable elimination algorithm?

Ignore variables that’s not an ancestor of the query variable or evidence variables. Because such variables is irrelevant to the query

#### Approximate inference

* + - * 1. Prior Sampling

The procedure generating a sample from the prior joint distribution. We can calculate the estimated probability of an event by calling this procedure N times, the estimated probability is the quotient of the number of samples that match the event and N

def prior\_sample(bn):  
 """Randomly sample from bn's full joint distribution. The result  
 is a {variable: value} dict. [Figure 14.13]"""  
 event = {}  
 for node in bn.nodes: # bn.nodes is ordered from parents to children  
 event[node.variable] = node.sample(event)  
 return event

The probability distribution from which the value is sampled is conditioned on the values already assigned to the variable’s parents, i.e. P(variable|variable’s parents)

* + - * 1. Rejection Sampling

An algorithm estimating the probability distribution of variable X given evidence e, by rejecting all samples that don’t match the evidence and counting the rest.

def rejection\_sampling(X, e, bn, N):  
 """Estimate the probability distribution of variable X given  
 evidence e in BayesNet bn, using N samples. [Figure 14.14]  
 Raises a ZeroDivisionError if all the N samples are rejected,  
 i.e., inconsistent with e.  
 >>> random.seed(47)  
 >>> rejection\_sampling('Burglary', dict(JohnCalls=T, MaryCalls=T),  
 ... burglary, 10000).show\_approx()  
 'False: 0.7, True: 0.3'  
 """  
 counts = {x: 0 for x in bn.variable\_values(X)} # bold N in [Figure 14.14]  
 for j in range(N):  
 sample = prior\_sample(bn) # boldface x in [Figure 14.14]  
 # count the samples that match the evidence e  
 if consistent\_with(sample, e):  
 counts[sample[X]] += 1  
 return ProbDist(X, counts)  
  
def consistent\_with(event, evidence):  
 """Is event consistent with the given evidence?"""  
 return all(evidence.get(k, v) == v  
 for k, v in event.items())

What’s the biggest problems with rejection sample algorithm?

It rejects too many samples. The fraction of samples consistent with the evidence e drops exponentially as the number of evidence variables grows, so the procedure is simply unusable for complex problems

* + - * 1. Likelihood Weighting

An algorithm estimating 𝑷(X|e) based to 𝑷(X|e)=a∑\_{y}𝑷(X, y, e), where a is normalization constant and y is the set of hidden variables, which is calculated by sampling

Algorithm Overview

def likelihood\_weighting(X, e, bn, N):  
 """Estimate the probability distribution of variable X given  
 evidence e in BayesNet bn. [Figure 14.15]  
 >>> random.seed(1017)  
 >>> likelihood\_weighting('Burglary', dict(JohnCalls=T, MaryCalls=T),  
 ... burglary, 10000).show\_approx()  
 'False: 0.702, True: 0.298'  
 """  
 W = {x: 0 for x in bn.variable\_values(X)}  
 for j in range(N):  
 sample, weight = weighted\_sample(bn, e) # boldface x, w in [Figure 14.15]  
 W[sample[X]] += weight  
 return ProbDist(X, W)  
  
def weighted\_sample(bn, e):  
 """Sample an event from bn that's consistent with the evidence e;  
 return the event and its weight, the likelihood that the event  
 accords to the evidence."""  
 w = 1  
 event = dict(e) # make a copy of e  
 for node in bn.nodes: # bn.nodes is ordered from parents to children  
 Xi = node.variable  
 if Xi in e:  
 w \*= node.p(e[Xi], event) # multiply w by P(e[Xi]|parents(e[Xi]) in event)  
 else:  
 event[Xi] = node.sample(event)  
 return event, w

What is the weight?

The likelihood that the event accords to the evidence, as measured by the product of the conditional probabilities of each evidence variable, given its parents

At first glance, it seems that weight\_sample always returns the same weight, w=∏node.p(e[Xi], event), for Xi in evidence variables, but it does not, because event might include different samples for different calls and node.p(e[Xi], event) is dependent on the values of e[Xi]’s parents in event

Why can we weigh events in that way?

We divide an event **x** into two part: one is evidence values **e** and the other is non-evidence value **z**, we have  
 The first part is calculated through samples and the second is the weight, the likelihood that the event accords with the evidence

What’s the advantage & disadvantage of likelihood weighting?

Advantage: It’s much more efficient than rejection sampling, because it uses all samples generated.

Disadvantage: It will suffer a degradation in accuracy as the number of evidence variables increases. This is because it will have very low weights and hence the weighted estimate will be dominated by the tiny fraction of samples whose weights are relatively high. The problem is exacerbated if the evidence variables occur late in the variable ordering, because then the non-evidence variables will have no evidence in their parents and ancestors to guide the generation of samples — all parents(e[Xi]) in event are sampled in previous steps instead of being indicated by the evidence. This means the samples will be simulations that bear little resemblance to the reality suggested by the evidence

* + - * 1. Gibbs sampling

A Markov chain Monte Carlo algorithm estimating 𝑷(X|e)

Algorithm Overview

The algorithm starts with an arbitrary state, and generates a next state by randomly sampling a value for a non-evidence variable Zi

def gibbs\_ask(X, e, bn, N):  
 """[Figure 14.16]"""  
 assert X not in e, "Query variable must be distinct from evidence"  
 # [value of X: times appearing in the samples]  
 counts = {x: 0 for x in bn.variable\_values(X)}  
 # the non-evidence variables in bn  
 Z = [var for var in bn.variables if var not in e]  
 # the current state of the network  
 state = dict(e) # boldface x in [Figure 14.16]  
 # initialize the state  
 for Zi in Z:  
 state[Zi] = random.choice(bn.variable\_values(Zi))  
 for j in range(N):  
 for Zi in Z:  
 # sampling a value for Zi according to P(Zi|MarkovBlanket(Zi))  
 state[Zi] = markov\_blanket\_sample(Zi, state, bn)  
 # this expression could be moved out of the current loop, IMHO  
 counts[state[X]] += 1  
 return ProbDist(X, counts)

Calculate 𝑷(X|X’s Markov blanket)

def markov\_blanket\_sample(X, e, bn):  
 """Return a sample from P(X | mb) where mb denotes that the  
 variables in the Markov blanket of X take their values from event  
 e (which must assign a value to each). The Markov blanket of X is  
 X's parents, children, and children's parents."""  
 Xnode = bn.variable\_node(X)  
 Q = ProbDist(X)  
 for xi in bn.variable\_values(X):  
 ei = extend(e, X, xi)  
 # [Equation 14.12: P(X|mb) = P(x|parents(X)) \* (P(ei[c]|parents(C)) for C in X's children)]  
 Q[xi] = Xnode.p(xi, e) \* product(Yj.p(ei[Yj.variable], ei)  
 for Yj in Xnode.children)  
 # (assuming a Boolean variable here)  
 return probability(Q.normalize()[True])

𝑷(X|mb) = a 𝑷(X|X’s parents) ∏\_{C ∈ X’s children}P(c|C’s parents), where a is the normalization constant

What’s the transition probability in the Gibbs sampling?

Because the algorithm changes the state by sampling a value for one non-evidence variable, Zi, the transition probability is the probability of sampling a value for Zi conditionally on all other variables—P(zi|~(Zi), e)—which, for Bayesian networks, is the probability conditionally on Zi’s Markov blanket—P(zi|Zi’s Markov blanket)

~(Zi) is all the non-evidence variable other than Zi

Why does Gibbs sampling work?

To show Gibbs sampling works, we need to prove that the sampling process has stationary distribution 𝑷(Z|e), which naturally follows if we can prove the transition probability—P(zi|zi’s Markov blanket), i.e. P(zi|~(Zi), e)—is in detailed balance with 𝑷(Z|e), now we show this:  
P(s) P(s, s’) = P(s|e) P(zi’|~(zi), e) = P(zi, ~(zi)|e) P(zi’|~(zi), e)  
 = P(zi|~(zi), e) P(~(zi)|e) P(zi’|~(zi), e)   
 = P(zi|~(zi), e) P(zi’, ~(zi)|e)  
 = P(s’) P(s’, s)

z is the values of non-evidence variables

zi is the value of non-evidence variable Zi

~(zi) is the values of non-evidence variables except that of Zi

Why is 𝑷(X|X’s Markov blanket) proportional to the probability of X given its parents times the probability of each child given its respective parent, i.e. 𝑷(X|X’s parents) ∏\_{C ∈ X’s children}P(c|C’s parents)?

First notice that 𝑷(X|X’s Markov blanket) = a ∏\_{C ∈ X’s children}P(c|C’s parents) 𝑷(X|X’s parents) ∏\_{Y ∈ X’s parents}P(y) ∏\_{Y ∈ children’s parents}P(y), Now analyze the expression term by term.   
𝑷(X|parents(X)) is definitely varies as X changes  
Because C’s parents includes X, P(c|parents(C)) varies as the value of X changes, so does ∏\_{C ∈ children(X)}P(c|parents(C))  
∏\_{Y ∈ parents(X)}P(y) and ∏\_{Z ∈ parents(C)}P(z) have nothing to do with X, so it can be incorporated into the normalization constant a  
Thus the above identity can be reduced to Proof completes.

Why does the algorithm iterate over Z, not just over X?

The difference emerges in markov\_blanket\_sample(Zi, state, bn), this function calculates 𝑷(Zi|Zi’s Markov blanket). If the algorithm only iterates over X, it implicitly regards all Zi except X as evidence variables, which, in turn, results in 𝑷(X|X’s Markov blanket) remaining the same all the time. And the answer the algorithm returns becomes approximate 𝑷(X|X’s Markov blanket), not 𝑷(X|e)

* + - * 1. Sampling summary

Prior sampling: To compute the approximate joint distribution, we sample from parents to children so that we can obtain P(X|parents(X)) while sampling

Rejection sampling: To compute the posterior distribution, we reject samples inconsistent with the evidence

Likelihood weighting: To avoid generate samples inconsistent with the evidence, we fix evidence variables and sample only hidden variables. In this way we need to assign a weight to each sample so that the resulting probability distribution is consistent

### Q & A

#### How to reduce the number of parameters needed to fill in a conditional probability table?

* + - * 1. Deterministic model

In which a node’s value is governed by its parents and the relationship between them, with no uncertainty (thus, the conditional probability is always 1)

Example:  
The relationship between parent nodes Canadian, US, Mexican and the child node NorthAmerican is simply that the child is the disjunction of the parents

* + - * 1. Noisy-OR

In which the probability that a node X is false given its parent’s value is the product of the inhibition probability for X’s parents which are true. More specifically, P(¬x|parents(X)) = ∏\_{i: X{i} ∈ Parents(X) ∧ X{i}=true}q{i}, where q{i} is the inhibition probability for X{i}. Thus P(x|parents(X)) = 1 - P(¬x|parents(X) = 1 - ∏\_{i: X{i} ∈ Parents(X) ∧ X{i}=true}q{i}

Inhibition

A parent Y is inhibited if it’s true but it’s child X is false, or more figuratively, Y is inhibited from causing X. The inhibition probability for Y is P(X=false|Y=true, other X’s parents are false}

Assumptions:  
1. All the possible causes are listed, if some are missing, we can always add a so-called leak node that covers "miscellaneous causes"  
2. Inhibition of each parent is independent of inhibition of any other parent. This makes sure we can compute something like P(X=false|Y=true, Z=true) by P(X=false|Y=true, Z=false)P(X=false|Y=false, Z=true). That means, to compute the full conditional probability table(CPT) for a node with k parents, which contains 2^k entries, we only need k inhibition probabilities, each for a parent.

Effect:  
Reduce the parameters needed to specify the conditional probability of a node with k parents from O(2^k) to O(k)

Example:  
Parents:   
 Cold, Flue, Malaria  
Child:   
 Fever  
Inhibition probability:  
 P(¬fever|cold, ¬flue, ¬malaria) = 0.6  
 P(¬fever|¬cold, flue, ¬malaria) = 0.2  
 P(¬fever|¬cold, ¬flue, malaria) = 0.1  
CPT:  
 P(fever|cold, flue, malaria) = 1 - 0.6 \* 0.2 \* 0.1 = 0.988  
 P(fever|cold, flue, ¬malaria) = 1 - 0.6 \* 0.2 = 0.88  
 …

#### What’s the standard deviation of the error in each probability?

* + - * 1. It’s proportional to 1/√(n), where n is the number of samples used in the estimate

#### How to add dependence between two variables?

* + - * 1. When those two variables is conditioned on their common child, they become conditionally dependent. Such an effect is called explaining away. As a simple example, considering following network:  
            Where S and R are independent. Things change when they are conditioned on H, i.e. P(S|H, R) ≠ P(S|H), even if P(S|R) = P(S). This is because R explains away the cause of H.

Explaining away results in one cause depending on another cause conditioned on their common effect

### TODO: 14.6

## Probabilistic Reasoning over Time

### Concept

#### Transition model / Transition probability

Which describes how the world involves. That is, the probability distribution of the variables at time t, given the state of the world at past time. i.e. 𝑷(X(t)|x(1: t-1)), where X(t) denotes the set of state variables at time t, x(1: t-1) denotes the set of state values from 1 to time t-1

#### Sensor model / Emission probability

Which describes how the evidence variable gets their values. That is, the probability of each percept at time t, given the current state of the world. i.e. 𝑷(e(t)|X(t)), where e(t) denotes the set of observable variables at time t, X(1: t) denotes the set of state values at time t

* + - * 1. Why do we need the sensor model?

The sensor model, i.e. 𝑷(e(t)|X(t)), describes the causal probability. With help of the sensor model and percepts observed, the agent will be able to update the belief state: now the agent infers the current state based not only on the transition model and the previous state, but also on the causal probability, P(e(t)|X(t)). Thus, the probability distribution of the belief state at time t, given the sensor model as well as the transition model, is  
 Where a is the normalizer.  
At the last step, we incorporate P(x(t-1)) into a, under the assumption that X(t-1)=x(t-1) and P(x(t-1)) is given before; And 𝑷(e(t)|X(t)) = 𝑷(e(t)|X(t), x(1:t-1)), assuming e(t) is independent of x(1:t-1) conditioned on X(t)

#### Markov Model

A stochastic model used to model randomly changing systems. It is assumed that future states depend only on the current state, not on the events that occurred before it

* + - * 1. Two representations

Which will be explored further when talking about [HMM](omnioutliner:///open?focus=knF9wZTuITE&row=kauzcZvPbqc)

Time-based

Xs are random variables whose values are states

State-based

A simple example with two states

* + - * 1. How to calculate the probability distribution of time t given the probability distribution of time t-1 in a Markov model?

* + - * 1. Does a Markov model always have a stationary distribution?

Yes, as explained [here](omnioutliner:///open?focus=m-j2eC1Kgrl&row=hq6_tEoWhEM)

#### Hidden Markov Model (HMM)

A statistical Markov Model in which the system being modeled is assumed to be a Markov process with unobserved (i.e. hidden) states.

* + - * 1. Hidden state

A discrete random variable which describes the state at some time slice

* + - * 1. Observation

An evidence variable, output of hidden states

* + - * 1. Transition model / Transition matrix

A square matrix, T, where T[i, j] = P(X{t}=j |X{t-1}=i)

* + - * 1. Sensor model / Emission matrix

An diagonal matrix, O, whose ith diagonal entry is P(e(t)|X(t)=i) and whose other entries are 0

* + - * 1. Two representations

Time-based

State-based

Arrows between states represents transition probabilities

Arrows between states and evidence are emission probabilities

The differences are:  
1. Nodes in the time-based representation are variables, while those in the state-based representation are actual values  
2. Arrows in the time-based representation are conditional probability tables, while those in the state-based representation are probabilities  
3. Because of 2, the time-based representation, in which the transition and emission probabilities could vary from time to time, is more suitable to make inference; the state-based representation, on the other hand, is usually used to learn transition and emission probabilities from a sequence of observations

* + - * 1. Inference task

All algorithms described below are based on the Markov property

Filtering

The task of computing the belief state — the posterior distribution over the most recent state — given all evidence to date, i.e. 𝑷(X(t)|e(1: t))

How to calculate the posterior distribution over the state at time t+1, given evidence to date t+1?

We split the evidence into two parts — one up to time t and the other at t+1.  
Intuitively  
1. We recursively compute the posterior distribution over time t conditioned on evidence up to time t.   
2. Then we use the transition model together with the posterior distribution just computed to obtain the predicted distribution over the state at time t+1  
3. At last, we use the sensor model to weigh the predicted distribution  
Mathematically, we have

The process of calculating 𝑷(X(t+1)|e(1:t+1)) is a memoryless process. The initial distribution 𝑷(X(0)|e(1:0)) = 𝑷(X(0))

We think of f(1:t)=𝑷(X(t)|e(1:t)) as a forward message

Relation between filtering and variable elimination

Running variable elimination with the variables in temporal order exactly mimics the operation of filtering

Prediction

The task of computing the posterior distribution over the future state, given all evidence to date, i.e. 𝑷(X(t+k)|e(1:t)), for some k > 0

How to compute the predicted distribution at time t+k+1 from evidence to time t?

Prediction can be viewed simply as filtering without the addition of new evidence.   
Thus intuitively, the predicted distribution at time t+k+1 is obtained only based on the transition model and the predicted distribution at time t+k  
The following recursive computation shows how to derive a prediction for t+k+1 from a prediction for t+k  
 Naturally, this computation still involves the transition model but the sensor model is gone

What will the predicted distribution become as time goes by?

The predicted distribution will eventually converge to a fixed point, which is the [stationary distribution](omnioutliner:///open?focus=m-j2eC1Kgrl&row=mtkwe5dE-ga) of the Markov process defined by the transition model

Mixing time

The time taken for the predicted distribution to reach the fixed point

What influences the mixing time?

The more uncertainty there is in the transition model, the shorter will be the mixing time and the more the future is obscured

Smoothing

The task of computing the posterior distribution over a past state, given all evidence up to the present, i.e. 𝑷(X(k|e(1:t)) for some k such that 0 ≤ k <t

How to compute the posterior distribution over a past state from evidence up to the present?

We split the evidence into two parts — one up to time k and the other from k+1 to t.  
Intuitively, the process of computing the posterior distribution over a past state consists of two parts: one is calculating the posterior distribution over the state at time k conditioned on evidence to time k, the other is calculating the distribution over evidence from k+1 to t conditioned on the state at time k  
Mathematically, we have  
 The first term is the result of filtering up to time k; the second term, which is viewed as a backward message, can be computed by a recursive process

Backward message

Now we explain the last part in above computation term by term: The first term, 𝑷(e(k+1)|x(k+1)), is the sensor model; the second term, P(e(k+2:t)|x(k+1)), is the recursive part, whose initial distribution 𝑷(e(t+1:t)|X(t)) are all 1 because e(t+1:t) is an empty sequence; The last term, 𝑷(x(k+1)|X(k)), is the transition model

Notice that the above process requires recalculate the all backward messages as time goes by

The most likely sequence

The task of computing the posterior distribution over a sequence of states that most likely generate a given sequence of observations, i.e. 𝑷(X(1:t)|e(1:t))

How to calculate the posterior distribution over a sequence of state that most likely generate a given sequence of observations to date t+1?

Viterbi algorithm

As we did in filtering, we split the evidence into two parts — one up to t and the other at t+1  
Intuitively, if we have the most likely sequence to each possible state x(t+1) at time t+1, then the maximum of them is the answer (the first maximum). Now the problem turns into computing the most likely sequence to each possible state x(t+1), which is similar to the shortest path problem — where the shortest path from s to d is the shortest path from s to some point v before d plus the cost from v to d. Here, the most likely sequence to x(t+1) is the most likely sequence to some state x(t) at time t such that, together with the transition probabilities and the emission probabilities, it yields the most likely sequence to x(t+1)  
Mathematically, we have  
 for each possible x(t+1)

This process is usually implemented by a dynamic programming algorithm called the Viterbi algorithm

Learning

The task to learn transition and sensor models from observations. The learning can be done as a by-product of smoothing.

Smoothing provides an estimate of what transitions actually occurred and of what states generated the sensor readings, and these estimation can be used to update the models.The updated models provide new estimates, and the process iterates to convergence.

The overall process is an instance of the expectation maximization or EM algorithm (TODO: Section20.3)

Randomly initialize the parameters (𝜃=π, A, B)​: initial state distribution, the initial transition probabilities, and the initial emission probabilities:

These parameters can also be set using prior information, which can speed up the algorithm and steer it toward the desired local maximum

Here assume the initial state is X(1) and is not given

Compute the forward message — the probability of being in state ​i at time t​ and having observation sequence y(1:t)​:  
 And the backward message — the probability of having the observation sequence y(t+1:T)​ given starting state i​ at time t​:

Calculate the probability of being in state i​ at time t​ given the observation sequence ​y(1:T)  
 And the probability of being state at ​i and j​ at time ​t and t+1​ given the observation sequence y(1:T)

Update the parameters:  
The initial state distribution  
 The transition probability:  
 The emission probability:

Iteratively repeat ii-iv until a desired level of convergence

#### Dynamic bayesian network (DBN)

A bayesian network, in which the values of variables at the current time step are only related to the values of variables in previous time steps and those in the current time step.

* + - * 1. State variable

Whose value is related to the values of variables in previous time steps

Transition probabilities are used to describe the posterior distribution for such variables

* + - * 1. Evidence variable

Whose value is related to the values of variables in the current time step

Emission probabilities are used to describe the posterior distribution for such variables

* + - * 1. Gaussian error model

In which the probability of error drops off as error increases

What’s the problem of a Gaussian error model?

A Gaussian error model suffers transient failure

* + - * 1. Transient failure / Sensor failure

Where the sensor occasionally decide to send some nonsense, i.e. observations significantly inconsistent with the corresponding states

How do transient failures cause problems for Gaussian error models?

If the probability of a large sensor error is significantly less than the probability to a transition to the error state, even if the latter is very unlikely, then the posterior distribution will assign a high probability to the error state.

How to deal with sensor failure?

To handle sensor failure, the sensor model must raise the probability of a sensor error to cover such a failure

* + - * 1. Transient failure model

Which copes with the transient failure by raising the probability of a sensor error

* + - * 1. An example for transient failure

Let’s consider a battery problem where both Battery (B for short) and BatteryMeter (BM for short) can take on discrete values 0 through 2. We further assume, in a Gaussian error model, the emission distribution, for BM=0, is 𝑷(BM=0|B)=[0.75, 0.2, 0.05]. Then the posterior distribution of the current state given BM=0 and B(t-1)=2, is 𝑷(B(t)|B(t-1)=2, BM=0)=[0.75, 0.2, 0.05], which is very high for B(t)=0], even if BM=0 is just a transient failure.  
A transient failure model, which considers the transient failure, raises the probabilities of sensor errors, so the emission distribution for, BM=0, may look like 𝑷(BM=0|B)=[0.6, 0.3, 0.2], and the posterior distribution is 𝑷(B(t)|B(t-1)=2, BM=0)=[0.6, 0.3, 0.2]

We assume the previous state is given here instead of being computed from previous evidence to simplify the computation for 𝑷(B(t)|B(t-1)=2, BM=0). This doesn’t effect the resulting conclusion: if we compute the state distribution from evidence, the resulting posterior distribution over B(t) will be dependent on both the transition model and the sensor model. But there is something not changed—in above example, 𝑷(B(t)=2|B(t-1)=2, BM=0), in the transient failure model, is 4 times than that in the Gaussian error model

For simplicity, timestamps are omitted in above example if not necessary

* + - * 1. Persistent failure

Where the sensor is broken

* + - * 1. Persistent failure model

Which introduces a new variable to indicate that the sensor is broken

* + - * 1. What’s the relation between HMM and DBN?

Every HMM is a single-variable DBN, every discrete DBN can be translated into an HMM.

When there’s only one state variable in DBN, it’s an HMM

To translate a DBN into a HMM, we combine all the state variables at a time slice into one mega-variable

* + - * 1. What’s the advantage of DBNs over HMMs?

By composing the state of a complex system into its constituent variables, CPTs in DBNs is much more sparse than those in HMMs

* + - * 1. Can a DBN be represented by a Kalman filter?

Not every DBN can. In a Kalman filter, the current state distribution is always a single multivariate Gaussian distribution — that is, a single “bump” in a particular location. DBNs, on the other hand, can model arbitrary distributions

* + - * 1. What information do we need to construct a DBN?

The prior distribution over the initial state variable, P(X(0))

The transition model

The sensor model

### Algorithm

#### Kalman filter

An algorithm that uses a series of measurements observed over time, containing [statistical noise](https://en.wikipedia.org/wiki/Statistical_noise) and other inaccuracies, and produces estimates of unknown variables that tend to be more accurate than those based on a single measurement alone, by estimating a joint probability distribution over the variables for each timeframe

* + - * 1. For the detailed mathematic computations, go check section 15.4. Away from the math for now, the essential idea behind this algorithm is, if the initial state distribution is a Gaussian, and the transition and sensor models are linear Gaussian, the predicted and updated state distributions are also a Gaussian. Furthermore, the variance update is independent of the observations, and thus, can be computed offline.

#### Particle filter

* + - * 1. First a population of N initial-state samples is created by sampling from the prior distribution P(X(0)). Then the update cycle is repeated for each time step:  
           1. Prediction: compute the next state value for each sample, x(t+1), given the current value, x(t), using the transition model, 𝑷(X(t+1)|x(t)).  
           2. Update: weigh each sample using the sensor model, P(e(t+1)|x(t+1)), and normalize them  
           3. Resample: select a new population of N samples from the current population. Each new sample is chosen from the current population in proportion to its weight

### A & Q

#### How to compute the likelihood of the evidence sequence P(e(1:t))?

* + - * 1. The last step borrows the computations from filtering, except that no normalizer is needed: notice that P(x(t) | e(1:t)) = aP(x(t), e(1:t))

## Sequential Decision Problems

### Markov Decision Process (MDP)

A sequential decision problem for a fully observable, stochastic environment with a Markov transition model and additive rewards.

#### Finite horizon

There’s, for the game, a time limit after which nothing matters

* + - * 1. The optimal policy for a finite horizon is nonstationary. That means the optimal action in a given state could change over time, in other word, it relatively depends on the deadline

#### Infinite horizon

There’s no time limit

* + - * 1. The optimal policy is stationary. It brings strong consequences: under stationarity there are just two coherent ways to assign utilities to state sequences

Additive rewards

, where R(s) is the short term reward for being in s

Discounted rewards

, where gamma is between 0 and 1. It is essential to realize that the agent has a preference for current rewards over future rewards

Large gamma suggests high weight to the future states, which means looking ahead further, and latter we’ll see it makes value iteration algorithm convergence slowly. Small gamma implies the agent is shortsighted

Additive vs Discounted

With discounted rewards, the utility of an infinite sequence is finite

If there is a proper policy that guarantees the agent to get to the terminal state, then we can use additive rewards

Even for finite sequence, additive rewards are still not equal to discounted rewards. Consider two sequences, [A, B, C, D] and [A, C, B, D] where R(B)>R(C). The utilities of these two sequences obtained with additive rewards are the same. With discounted rewards, however, U([A, B, C, D]) > U([A, C, B, D])

#### Problem description

* + - * 1. (S, T, A, R, \gamma)

S: states

T: transition model

A: actions

R: rewards

\gamma: discount factor

#### Policy & Utility

Policy specifies what the agent should do for any state

Utility specifies the total rewards from a state onward.

* + - * 1. The optimal policy are strongly correlated with state utilities: with one known, the other naturally follows

Given a policy pi, the expected utility for state s  
 It can be calculated recursively by   
 Where s’ is the possible next state by executing \pi(s) at s

The above recursive equation can be thought as a linear programming problem

Given utilities for each state, the optimal policy is to choose an action a which maximizes the expected utility from s onward

Note the optimal policy varies according to utilities, which in turns depends on reward. That implies a change of reward will affect the optimal policy

### Partially Observable Markov Decision Process (POMDP)

### Algorithm

#### MDP

* + - * 1. Value iteration

Algorithm

def value\_iteration(mdp, epsilon=0.001):  
 """  
 param: epsilon, the maximum error allowed in the utility of any state  
 """  
 U1 = {s: 0 for s in mdp.states}  
 # reward function, transition model, discount factor  
 reward, transition, gamma = mdp.R, mdp.T, mdp.gamma  
 while True:  
 U = U1.copy()  
 # delta: maximum change in the utility of any state in an iteration  
 delta = 0  
 for s in mdp.states:  
 U1[s] = reward(s) + gamma \* max([sum([p \* U[s1] for (p, s1) in transition(s, a)])  
 for a in mdp.actions(s)])  
 delta = max(delta, abs(U1[s] - U[s]))  
 # return the utility when the change in utility between consecutive iterative is very small  
 if delta < epsilon \* (1 - gamma) / gamma:  
 return U

Algorithm Description

U1 = initial utility function

while the utility function has a considerably large change between consecutive iteration:

U = a copy of U1

delta = 0

for each state s:

use the current utility function to calculate a new utility U1(s) for the state s

update delta if the difference between the newly calculated utility U1(s) and the old one U(s) is greater than delta

return U if delta is small

Why do we restrain delta less than epsilon \* (1-gamma)/gamma?

It is said that   
If ||U{i+1}-U{i}|| < epsilon(1-gamma)/gamma  
Then ||U{i+1}-U|| < epsilon

i.e. if the update between consecutive iterations is small, then the error between estimate and the true utility function is also small

In the above process, the policy \pi becomes optimal long before U{i} has converged

* + - * 1. Policy iteration

Which is based on the insight that the policy \pi becomes optimal long before U{i} has converged

Algorithm

def policy\_iteration(mdp):  
 # initialization  
 U = {s: 0 for s in mdp.states}  
 pi = {s: random.choice(mdp.actions(s)) for s in mdp.states}  
   
 while True:  
 U = policy\_evaluation(pi, U, mdp)  
 unchanged = True  
 # policy improvement  
 for s in mdp.states:  
 a = argmax(mdp.actions(s), key=lambda a: expected\_utility(a, s, U, mdp))  
 if a != pi[s]:  
 pi[s] = a  
 unchanged = False  
 if unchanged:  
 return pi  
  
def expected\_utility(a, s, U, mdp):  
 """The expected utility of doing a in state s, according to the MDP and U."""  
 return sum([p \* U[s1] for (p, s1) in mdp.T(s, a)])

Algorithm Description

initialize utility function U and policy pi

do

policy evaluation: update U according to the current policy pi

policy improvement: improve the policy pi based on the updated U

until pi isn’t changed

Policy evaluation

def policy\_evaluation(pi, U, mdp, k=20):  
 """Return an updated utility mapping U1 from each state in the MDP to its  
 utility, using an approximation (modified policy iteration)."""  
 # reward function, transition model, discount factor  
 R, T, gamma = mdp.R, mdp.T, mdp.gamma  
 U1 = U.copy()  
 for i in range(k):  
 for s in mdp.states:  
 U1[s] = R(s) + gamma \* sum([p \* U[s1] for (p, s1) in T(s, pi[s])])  
 return U1

Math behind the above algorithm:

Heads up: The above step need to repeat k times to get a good approximation. An exact policy evaluation return the utility when it doesn’t change or when it’s smaller than some tiny constant.

Strictly speaking, policy evaluation need to get to the equilibrium  
 It’s time-consuming (O(n^3)) to do the exact policy evaluation. Instead, the approximation given in the above implementation works fine when dealing with large state space

In this case, it’s almost the same as value iteration except that policy evaluation has a policy set in stone, which eliminates max in value iteration

* + - * 1. Relation between value iteration and policy iteration

If we modify value iteration so that it stops when the greedy policy doesn’t change and set to 1 k in policy evaluation for policy iteration, then value iteration is somewhat identical to policy iteration

* + - * 1. The algorithms we have described so far require updating the utility or policy for all states at once. It turns out that this is not strictly necessary. In fact, on each iteration, we can pick any subset of states and apply either kind of updating (policy improvement or sim- plified value iteration) to that subset. This very general algorithm is called asynchronous policy iteration. Given certain conditions on the initial policy and initial utility function, asynchronous policy iteration is guaranteed to converge to an optimal policy. The freedom to choose any states to work on means that we can design much more efficient heuristic algorithms—for example, algorithms that concentrate on updating the values of states that are likely to be reached by a good policy. This makes a lot of sense in real life: if one has no intention of throwing oneself off a cliff, one should not spend time worrying about the exact value of the resulting states.

## Learning from Examples

## Knowledge in Learning

## Learning Probabilistic Models

## Reinforcement Learning

### Concept

#### Credit assignment problem

How do you distribute credit for success among the many decisions that may have been involved in producing it?

#### Bellman equation

* + - * 1. Four equivalent version

U and V are the maximum estimated utility the agent could obtain from s onward

C is the maximum estimated utility the agent could obtain after executing a on s

Q is the maximum estimated utility the agent could obtain from s onward by taking action a

#### Passive reinforcement learning

In which the agent executes a set of trials in the environment using a fixed policy and learn a utility function on states

#### Active reinforcement learning

In which the agent must learn the model and decide what actions to take on its own

* + - * 1. Exploitation-exploration tradeoff

The agent must make a tradeoff between exploitation and exploration. The former one maximizes its reward, whereas the latter one helps the agent learn the environment and thereby contribute to its long-term well-being

#### Maximum likelihood estimation

Which estimates the parameter of a model so as to maximize the likelihood of observations

#### Reward

* + - * 1. Why might we want to change the reward function?

Make the problem more efficient to solve, in both time and space

Make the problem solvable

* + - * 1. Reward-modification strategy

Multiplying by a scalar, i.e. R’ = cR, and the resulting Q’ = cQ

Adding a scalar, i.e. R’ = R + c, and the resulting Q’ = Q + c/(1-gamma)

* + - * 1. Potential-based (reward) shaping

We rewrite R as below  
 And now Q becomes

Phi(s) is a shaping function, which maps each state to a real value (potential)

The result Q’(s, a) = Q(s, a) - \Phi(s) is quite intuitive that since we map each state to a value, we remove the value of the state from our estimated utility when we leave a state

Note, potential-based shaping doesn’t change the original policy

Why do we define potential other than reward?

Reward is accumulated along the way, which doesn’t get diminished (unless the agent receives negative reward).  
Potential, on the other hand, is the temporal reward the agent receives when entering a state, which will be removed when the agent leaves the state

Potentials are based on the knowledge of states in the environment. They help speed up reinforcement learning. The effect of potential-based shaping is equivalent to initializing the Q-value with potentials.

In a tabular setting with a fixed potential function, initialization is feasible and might be simpler. But it gets tricker with function approximation and impossible with potential functions that change over time

Potential-based reward shaping still provides the only Bellman-consistent mechanism of combining value functions

### Algorithm

#### Model-based

* + - * 2. Adaptive dynamic programming

Algorithm Overview

Further modifications are required to work with exploration, refer to Q-learning for an example

def ADP(s’, r’) -> action  
 if s’ is new   
 U[s’] = r’  
 R[s’] = r’  
 if s is not null  
 ++N1[s, a, s’]  
 ++N2[s, a] # update transition model  
 for each t such that N[s, a, t] is not zero  
 P(t|s, a) = N1[s, a, t] / N2[s, a] # a procedure of MLE  
 update\_utility(U)  
 If s’ is not a terminal state  
 s = s’  
 a = select\_action()  
 return a

s: previous state

a: previous action

s’: current state

r’: reward for state s’

U: utility

R: reward

N1: frequencies for state-action-successors

N2: frequencies for state-actions

P: transition probabilities

Utility update

In passive RL, where a fixed policy is given, we update utilities as follows

In active RL, where the agent has to make a tradeoff between exploitation and exploration, utilities are updated as follows  
 discussion of f is left [here](omnioutliner:///open?focus=bquDZh2B8mI&row=me3OFZqmiVF)

Action selection

In passive RL, the action is specified by the fixed policy

In active RL, it’s the action that maximizes the expected reward-to-go

Exploration function f

Which determines how exploitation is trade off against exploration

f(u, n) should be increasing in u and decreasing in n. A simple definition is  
 It says if n is less than a fixed parameter N, return the best possible reward R (∀u R≥u), otherwise return u

How to update utility efficiently?

The prioritized sweeping heuristic only adjusts states whose successors have just undergone a large adjustment. That is, U(s) gets updated only when U(s’) has changed much, where s’ is some successor of s

#### Model-free

* + - * 2. Direct utility estimation

For each trial, it backward computes reward-to-go for each state on the way and updates the estimated utility for that state accordingly

Algorithm Overview

For each trial t for s in reverse(state in t)  
 U(s) = U(s) + 1/t \* (U\_{t}(s) - U(s))

U(s): utility estimate for state s

U\_{t}(s): reward-to-go for state s in trial t

Drawbacks: it updates utility for each state individually and neglects the fact that the utilities of states are dependent

* + - * 1. Temporal-difference learning based on utility

In which utility is updated simply by the difference in utilities between successive states

Algorithm Overview

Further modifications are required to work with exploration, refer to Q-learning for an example

def TD(s’, r’) -> action  
 if s’ is new:  
 U[s’] = r’  
 if s is not null  
 ++N[s]  
 U[s] = U[s] + alpha(N[s])(r + gamma \* U[s’] - U[s])  
 if s’ is not a terminal state  
 s = s’  
 a = select\_action()  
 r = r’  
 return a

s: previous state

r: reward for state s

s’: current state

U: utility

N: frequencies of s

gamma: discount factor

Learning rate alpha(t)

Which determines to what extent the newly acquired information overrides old information

The sum of alpha(t) should be infinite

This suggests that if alpha(t) = (1/t)^n, then n <= 1

The sum of squared alpha(t) should be less than infinite

This suggests that if alpha(t)^2 = (1/t)^(2n), then 2n > 1

Utility update

r + gamma \* U[s’] is the inferred utility for s with known utility for its successor s’. Thus, TD updates the utility for previous state s by adding difference between the inferred utility for s and the original utility for s

The way TD updates utility is analogous to that of direct utility estimation.

Action selection

In passive RL, the action is specified by the fixed policy

In active RL, it’s the action that maximizes the expected reward-to-go

In order to obtain P(s’|s, a), we may still need to do some extra work as we did in ADP. And that makes TD model-based

Why do we say TD is a combination of Monte Carlo method and Dynamic Programming?

It’s a Monte Carlo method because the next state is randomly sampled based on the transition probability

It’s a dynamic programming because the update step is based on previously computed values

* + - * 1. Q-learning

An alternative TD method based on the Qualities of actions, in which no transition model is involved

Algorithm Overview

def Q-learning(s’, r’) -> action  
 if s is terminal  
 Q[s, None] = r’  
 if s is not null:  
 ++N[s, a]  
 Q[s, a] = Q[s, a] + alpha(N[s, a])(r + gamma \* max\_{a’}(Q[s’, a’]) - Q[s, a])  
 s = s’  
 a = select\_action  
 r = r’  
 return a

Action selection

The action with maximum quality

Such policy is called the behavioral policy

What makes Q-learning model-free?

The TD approach helps cut off the dependency of transition model when updating Q. By transforming bellman equation from state-utility to action-utility, it makes action selection independent of transition model

How to update theta when we use function approximator to approximate Q-function?

Define function approximator for Q-function  
 And loss function  
 We update each theta{i} every time Q-learning is invoked

* + - * 1. SARSA(state-action-reward-state-action)

A similar algorithm to Q-learning

What’s the difference between Q-learning and SARSA?

Q-learning is an off-policy algorithm, whereas SARSA is an on policy algorithm

SARSA requires one more information than Q-learning — action a’ executed at state s’. That means SARSA updates Q(s, a) after a’ is actually taken, while Q-learning update Q(s, a) after a is taken. With help of a’, SARSA updates Q as below

For reference, update equation Q-learning uses to update Q is

Both works identically when no exploration happens. When exploration is happening, however, they differ significantly. Because Q-learning uses the best Q-value in the update rule, it, unlike SARSA, neglects the actual action being taken. Q-learning is more flexible, in the sense that a Q-learning agent can learn how to behave well even when guided by a random or adversarial exploration policy. On the other hand, SARSA is more realistic, for example, if the overall policy is even partly controlled by other agents, it’s better to learn a Q-function for what will actually happen rather than what the agent would like to happen

#### Policy search

* + - * 1. What’s the policy representation used in policy search methods, and why?

Instead of using  
 Policy search methods often use a stochastic policy representation  
 In this way it introduces continuity into policy and thereby avoids an infinitesimal change in theta causing the policy to switch from one action to another

* + - * 1. How to improve policy?

Define policy value  
 where R(a) is the reward received after executing action a  
To improve policy, we tweak theta around according to the policy gradient vector   
 stop when policy doesn’t improve.

It’s more complicated in a stochastic environment, since R(a) varies from trial to trial. In that case, we can do N trials in advance, then we have the policy gradient vector  
 Where a{j} is the action executed in s on the jth trial, R(a{j}) is the reward received by taking a{j} in the jth trial

### Q & A

#### What’s a state?

* + - * 1. A state is a function of history

#### What’s the difference between RL and MDP?

* + - * 1. RL lacks information about the environments, it has to explore the environment in order to learn the MDP

#### What makes reinforcement learning different from other machine learning paradigms?

* + - * 1. There is no supervisor, only a reward signal
        2. Feedback is delayed, not instantaneous
        3. Time really matters (sequential, non i.i.d. data)
        4. Agent’s actions affects the subsequent data it receives

#### Comparison between ADP and TD

* + - * 1. TD adjusts of a state to agree with its observed successor, whereas ADP adjusts the state to agree with all of the successor that might occur, weighted by their probabilities. This difference disappears when the effects of TD adjustments are averaged over a large number of transitions, because the frequency of each successor in the set of transitions is approximately proportional to its probability
        2. TD is more efficient in terms of computation, where ADP converges more quickly

#### Comparison between model-based and model-free agents

* + - * 1. Knowledge-based agents work better, especially in complex environment, than model-free agents.
        2. Model-based algorithm becomes impractical as the state space and action space grows (S\*S\*A, for a tabular setup). On the other hand, model-free algorithms rely on trial-and-error to update its knowledge. As a result, it does not require space to store all the combination of states and actions

#### Why do we bother to use function approximation to represent value function?

* + - * 1. Traditional table-based approach is inefficient and even ineffective to handle the case where the state space is huge
        2. A function approximation allows agent to generalize (which Q-learning can’t do) from visited states to the unvisited.

#### What kind of function approximations are not suitable for reinforcement learning?

* + - * 1. Reinforcement learning is known to be unstable or even diverge when a nonlinear function approximator such as neural network is used to represent Q-function.

#### Why is some function approximator not suitable for reinforcement learning?

* + - * 1. The correlations present in the sequence of observations

Solution: experience replay

* + - * 1. The fact that small updates to Q may significantly change the policy and therefore change the data distribution

Solution experience replay

* + - * 1. The correlations between the action-value and the target values

Solution: use a separate target network to calculate the target value

#### What’s the difference between on-policy and off-policy algorithms?

* + - * 1. On-policy algorithms learn value functions based on the current policy, whereas off-policy algorithms learn it based on another policy (greedy policy for Q-learning)

#### What’s the end-to-end reinforcement learning?

* + - * 1. An end-to-end learning use a single neural network to build connections between input and output without further modularizing the process into different parts. For example, traditional computer vision pipeline modularize the processing into image preprocessing, interesting area selection, feature extraction, etc; where an end-to-end learning algorithms like CNN build direct bridge between input and output.

#### What’s the advantage/disadvantage of policy-based RL against value-based RL?

* + - * 1. Advantages

Better convergence properties: value-based methods can oscillate even diverges in some cases, where policy-based methods are guaranteed to converge at least to a local optimum if we just directly follow the policy gradient

Effective in high-dimensional or continuous action spaces: value-based methods require a maximization operation to select an action at each step. This operation could be intimidating when action space is large. Policy-based methods circumvent this operation by adjusting the parameters of the policy directly, and thereby incrementally learn what’s the best action to take at each state

Can learn stochastic policy

* + - * 1. Disadvantages

Naive policy-based methods converge very slowly, and have high variance.

#### Why not separate exploration and exploitation completely, i.e. do exploration during the training, and do exploitation after the training?

* + - * 1. Because the agent is learning online.

### TODO

#### Why do we bother to learn the above algorithms since we can simply use function approximation to learn utility/Q- function?

#### P847

# Problems

## Sudoku

The original paper is [here](http://norvig.com/sudoku.html)

### Preliminary Notations

#### Label columns 1-9, the rows A-I, so that squares can be represented as A1, A2, …, B1, B2, …, …, I1, I2, …

#### A unit is a collection of 9 squares (column, row, or box)

#### The peers of a square is the squares that share a unit with that square. So there are 20 peers for each square.

### Two Strategies that we can use to make process towards filling all the squares

#### If a square has only one possible value, then eliminate that value from the square’s peers

#### if a unit has only one possible place for a value, then put the value there

### Solution 1.

#### Initiate a map in which every square can be any digit

#### Assign the known values to the corresponding squares, respectively.

i.e., eliminate all other possible digits from that square

* + - * 1. During the eliminations, we check whether our two strategies can be applied

#### Using depth-first search to find the ultimate answer

Choose the square with fewest choices.

There is one thing notable: when we randomly assign a value to a square, call assign defined before

#### Python Code

* + - * 1. """  
           Constants  
           """  
           digits = '123456789'  
           rows = 'ABCDEFGHI'  
           cols = digits  
           def cross(a, b):  
            return [s+t for s in a for t in b]  
           squares = cross(rows, cols)  
           row\_units = [cross(r, cols) for r in rows]  
           column\_units = [cross(rows, c) for c in cols]  
           square\_units = [cross(rs, cs) for rs in ('ABC','DEF','GHI') for cs in ('123','456','789')]  
           unitlist = row\_units + column\_units + square\_units  
           units = dict((s, [u for u in unitlist if s in u]) for s in squares)  
           peers = dict((s, set(sum(units[s],[]))-set([s])) for s in squares)  
           blank\_map = dict((s, digits) for s in squares)  
             
             
           def solve(sudoku\_string):  
            """  
            Args:  
            sudoku\_string: the string representation of the given sudoku map, comprised of 1-9, if the value is given, and 0 or \*, if not  
            Return:  
            the dictionary form of a complete sudoku map if there is  
            False if a contradiction is detected  
            """  
            def assign(sudoku\_map, square, digit):  
            """Assign digit to sudoku\_map[square], eliminate all other possible digits from that square  
            Return:  
            modified sudoku\_map if assignment proceeds successfully  
            False if a contradiction is detected  
            """  
            other\_values = sudoku\_map[square].replace(digit, '')  
            if all(eliminate(sudoku\_map, square, d) for d in other\_values):  
            return sudoku\_map  
            else:  
            return False  
              
              
            def eliminate(sudoku\_map, square, digit):  
            """Eliminate digit from sudoku\_map[square]  
            Return:  
            modified sudoku\_map if elimination proceeds successfully  
            False if a contradiction is detected,  
              
            """  
            if digit not in sudoku\_map[square]:  
            return sudoku\_map ## already eliminated  
              
            sudoku\_map[square] = sudoku\_map[square].replace(digit, '')  
              
            if len(sudoku\_map[square]) == 0:  
            return False ## contradiction: remove the last value  
            elif len(sudoku\_map[square]) == 1:  
            ## current square has only one possible value, put that value here and eliminate that value from all its peers  
            if not all(eliminate(sudoku\_map, s, sudoku\_map[square]) for s in peers[square]):  
            return False  
              
            ## check whether a unit has only one possible place for digit  
            for unit in units[square]:  
            squares\_contain\_digit = [s for s in unit if digit in sudoku\_map[s]]  
            if len(squares\_contain\_digit) == 0:  
            return False ## contradiction: no place for digit  
            elif len(squares\_contain\_digit) == 1:  
            ## current unit has only one possible for digit, put it here  
            if not assign(sudoku\_map, squares\_contain\_digit[0], digit):  
            return False  
            return sudoku\_map  
              
            def initiate\_sudoku\_map(sudoku\_string):  
            """Convert the string representation of sudoku to dictionary one: {square: digit}"""  
            assert(len(sudoku\_string) == 81)  
            return dict(zip(squares, sudoku\_string))  
              
              
            def search(sudoku\_map):  
            """Using depth-first search to find the ultimate answer  
            Args:  
            sudoku\_map(dict): a dictionary of the form {square: digit, ...}  
            Return:  
            the dictionary form of a complete sudoku map if there is  
            False if a contradiction is detected  
            """  
            if sudoku\_map is False:  
            return False ## Failed earlier  
            if all(len(sudoku\_map[s]) == 1 for s in squares):  
            return sudoku\_map ## solved  
              
            ## choose the unfilled square with the fewest choices  
            choices, square = min((len(sudoku\_map[s]), s) for s in squares if len(sudoku\_map[s]) > 1)  
            for d in sudoku\_map[square]:  
            new\_sudoku\_map = sudoku\_map.copy()  
            assign(new\_sudoku\_map, square, d)  
            result = search(new\_sudoku\_map)  
            if result:  
            return result  
            return False  
              
            sudoku\_map = blank\_map.copy()  
              
            for square, digit in initiate\_sudoku\_map(sudoku\_string).items():  
            if digit in digits and not assign(sudoku\_map, square, digit):  
            return False  
              
            return search(sudoku\_map)  
           def display(sudoku\_map):  
            """Display sudoku map in 2-D form"""  
            width = 1+max(len(sudoku\_map[s]) for s in squares)  
            line = '+'.join(['-'\*width\*3]\*3)  
            for r in rows:  
            print(''.join(sudoku\_map[r+c].center(width) + ('|' if c in '36' else '') for c in cols))  
              
            if r in 'CF':  
            print(line)  
            print()  
              
           diag\_sudoku\_grid = '2.............62....1....7...6..8...3...9...7...6..4...4....8....52.............3'  
           display(solve(diag\_sudoku\_grid))

### Solution 2.

#### Initiate a map with given values

#### Iteratively apply two strategies until there is no more progress

ie. After two strategies applied, the number of squares with only one digit stay unchanged.

#### Using depth-first search to find the ultimate answer

At the beginning of search, above step is applied. Furthermore, unlike solution 1, assign operation is normal.