CS156 – Introduction to Artificial Intelligence Final Exam Review

By: Zayd Hammoudeh

Introduction to Agents

An agent perceives its **environment** through **sensors** and acts upon the environment through **actuators**.

Turing Test – A test where a human poses a series of questions to the computer and after seeing the responses cannot distinguish the **responses** from those of a human. **Components Needs to Pass the Turing Test:** 1. **Natural Language Processing** 2. **Knowledge Representation** (i.e. storage paradigm) 3. **Automated Reasoning** 4. **Machine Learning**	**Total Turing Test** – A variant of the Turing Test where the robot passes entirely as a human. **Additional Requirements Over Standard Turing Test:** 1. **Computer Vision** 2. **Robotics**	**Rational Agent** – For **every possible percept sequence**, the rational agent selects the action it expects to **maximize its performance measure** given the information in the **percept sequence** and whatever **built-in knowledge** it has. **The maximizing action depends on:** 1. **Performance Measure** 2. **Any prior/built-in knowledge of the agent** 3. **Percept sequence to date.** 4. **Set of possible actions.**	**Percept** – An agent's perceptual inputs through **sensors** at any given instant. **Percept Sequence** – Set of all percepts to date.

Agent Function: **Map from percept sequences to an agent action**. **Example:** An agent action table.

Agents run an agent program. The agent program runs on the **agent architecture**. The combination of the **agent program** and agent architecture is called a **complete agent**.

Cognitive Science: Brings together computer models from AI and experimental techniques from psychology to construct precise and testable theories of the human mind.

Task Environment (PEAS)

Performance Measure (P) – Targets/goals the agent will try to achieve.	**Environment (E)** – Objects that interact with the agent or the agent interacts with	**Actuators (A)** – Tool(s) used by the agent to interact with the environment.	**Sensors (S)** – Tool(s) used by the agent to perceive the environment.

Properties of a Task Environment

Fully Observable vs. Partially Observable Can the agent see the entire environment **at once** (e.g. chess)? If not, it may keep a history of what it has observed (taxi-driver).	**Deterministic vs. Stochastic** Is the next state completely determined by the current state and the action (chess)? Otherwise it is stochastic (taxi-driver).	**Single-Agent vs. Multi-agent** Do objects in the environment need to be treated as other agents? Multi-agent environments can be **competitive** (chess) or **cooperative** (taxi-driving). **Communication** between agents is possible as is **randomized behavior** to avoid predictability.	**Episodic vs. Sequential** In an episodic environment, the agent's experience is divided into episodes. In an episode the agent receives **one percept** and performs **one action** (e.g. quality control robot). In sequential environments, **current actions affect future actions**.
Static vs. Dynamic Does the environment **change while the agent is making a decision**? Chess is static while taxi driving is dynamic.	**Discrete vs. Continuous** Time, percepts, and actions divided **into a fixed, finite set** (e.g. chess)? A continuous environment is taxi-driving.	**Known versus Unknown** In a known environment, all outcomes of actions are known. In an unknown environment, the agent needs to figure out how it works to make good decisions.	

Example Episodic Agent

Quality Assurance robot.

- **Performance Measure:** Fixed minimum and maximum tolerances for a widget. (Example ball board min/max weight, diameter, roundess)
- **Environment:** Widget (example ball bearing) received for inspection on an input system. Good bin and discard bins.
- **Actuator:** Arm to place widget in either discard bin or good bin.
- **Sensor:** Check ball bearing weight, diameter, roundness etc.

Types of Agent Programs

Simple Reflex Agent – Select actions based off the **current percept only.** Often defined by **condition-action rules** (i.e. **productions**)	**Model-Based Reflex Agent** – Similar to a Finite State Automata. Uses **internal states** to keep track of the environment. Updates the internal state based off how the environment evolves independently and how the agent's action affect the environment. This is called the agent **model**.
Goal Based Agents – A **goal** is a binary condition (i.e. either met or not met). A goal based agent tries to reach a target goal. **Search and planning agents** may be goal based agents.	**Utility Based Agent** – Agent applies a **utility function** to its performance. Agent tries to maximize its overall utility function.

Additional Definitions

Problem solving agents deal with **atomic environments** (i.e. the environment is treated as a single whole and is **indivisible**).	**Planning agents** deal with **factored or structured environments** (i.e. the environment has **attributes/variables** each of which has a **value**).	**Search** – Process of looking over a sequence of actions.	**Solution** – A sequence of actions that takes the agent from the initial state to the goal state.

Search Problems

Classical search problems are **deterministic**, **fully-observable**, **known**, and the solution is a **sequence of actions**.

Solution: A sequence of actions that takes the agent from the initial state to the goal state.	**Root**: Initial State **Edge/Branches**: Actions **Node/Vertices**: States in the state space **Leaf**: A node with no children	**Node Expansion** – Applying all legal actions to the node and **generating** all successor states.	**Frontier or Open List** – Set of successor nodes that have not yet been expanded.
Search Strategy: Method for choosing the node on the frontier to next expand.	**Repeated State:** Any state visited more than once during a search. **Redundant Path:** Any two or more paths that go to the same state.	**Closed or Explored Set:** States that have already been expanded.	**Loopy Path** – Where a repeated state is expanded causing you not to continue to explore the same section of a graph.

Definitions:

Uniformed Search – Also known as (**Blind Search**) is any search that has no information on the search space.	**Informed Search** – Uses **heuristics** that inspect the state space to prioritize moves.	**Explored Set** – Set of all nodes already visited.
Branching Factor (*b*) – Number of branches/children/successors from a given node. Generally lists as the **maximum branching factor.**	**Depth (*d*)** – Number of branches/children/successors from a given node.	**Frontier Set** – Set of all nodes available for expansion.

A Problem consists of five attributes:

1. **Initial State**
2. **Set of possible actions** (ACTIONS)
3. **Successor Function/Transitional Model** (RESULTS)
4. **Goal test** (TERMINAL-TEST)
5. **Cost Function**

Four Ways to Rate/Measure a Search Strategy:

1. **Completeness** – If a solution exists, does the algorithm always find it?
2. **Optimal** – Is the solution found by the algorithm always optimal (i.e. have the lowest cost).
3. **Time Complexity** – Amount of time required by the algorithm to perform the search.
4. **Space Complexity** – Amount of memory required by the algorithm to perform the search.

Name	Memory Complexity	Time Complexity	Complete	Optimal	Queue Type Used	Comments
Depth Limited Search	$O(l)$	$O(b^l)$	No	No	Stack	l is the maximum allowed depth. 1. Incomplete if $d > l$ 2. Can be non-optimal if $l > d$
Depth-First Search	$O(d)$	$O(b^d)$	Yes if the graph is finite, No otherwise	No	Stack	1. Not complete because of the infinite branching problem (e.g. loop). 2. Can be considered special case of depth-limited search with $l = \infty$ **Always expand left most node that can be expanded.**
Iterative Deepening Depth First Search	$O(d)$	$O(b) + O(b^2) + \cdots + O(b^d) = O(b^{d+1})$	Yes	Yes	Stack	Calls Depth Limited Search algorithm d times
Breadth First Search	$O(b^d)$	$O(b^d)$	Yes	Yes if uniform step cost	Queue	Can be considered a variant of uniform cost search where each step cost is the same. **Expand the root node and then expand all children of the root node in the order they are encountered until all nodes are expanded or a goal is reached.**
Bidirectional Search	$O\left(b^{\frac{d}{2}}\right)$	$O\left(b^{\frac{d}{2}}\right)$	Yes	Yes if uniform step cost	Queue	Variant of Breadth-First Search where two breadth first searches (one from start and one from the goal) are initiated and carried out simultaneously. **Generalization of Breadth-First where the root (i.e. initiate state) node is expanded first and nodes are expanded based of their non-decreasing distance/cost from the root.**
Uniform Cost Search	$O\left(b^{1+\frac{C^*}{\epsilon}}\right)$	$O\left(b^{1+\frac{C^*}{\epsilon}}\right)$	Yes	Yes	Priority Queue	Variant of Breadth-First Search where the step cost is not uniform. C^* - Minimum (optimal) cost to the goal. ϵ - Minimum step cost
Greedy Best First Search	N/A	N/A	No	No	None	Selects node for expansion based off the one with the **lowest heuristic cost**. $f(n) = h(n)$ Can oscillate in a dead end condition.
A*	Based off quality of heuristic	Based off quality of heuristic	Yes	Yes with heuristic conditions	Priority Queue	
Recursive Best First Search	$O(d)$	Based off quality of heuristic	Yes	Yes if heuristic admissible	Stack	

Completeness above assumes the branching factor is **finite**.

Iterative Deepening Depth First Search (also known as Iterative Lengthening Search)

```
def ID_DFS(problem, limit):
    # Incrementally increase the maximum depth
    for maximum_depth in range(0, limit):
        result = Depth_Limited_Search(problem.INITIAL_STATE(),
                                      problem, maximum_depth)
        # If solution found return it.
        if(result is not None):
            return result
```

```
def Depth_Limited_Search(node, problem, depth):
    if(problem.GOAL_TEST(node)):
        return SOLUTION(node)
    if(depth == 0):
        return None
    for action in problem.ACTIONS(node):
        child = problem.RESULT(node, action)
        result = Depth_Limited_Search(child, problem, depth – 1)
        if(result is not None):
            return result
    return None
```

Space Complexity: $O(d)$ since at one time only keeping in memory at most d nodes.	**Time Complexity:** Depth-Limited-Search is called up to d times. Each call to Depth-Limited-Search takes $O(b^m)$ time. Given: $\sum_{i=m}^{n-1} a^i = \frac{a^m - a^n}{1-a}$, Then $b + b^2 + b^3 + \cdots + b^d = O(b^{d+1})$	**Complete:** Yes since all nodes are explored if $d \leq limit$	**Optimal:** Yes if all steps have uniform cost.

Uniform Cost Search (Uniformed Search)

Uniform cost search explores nodes on the frontier based of a monotonically increase cost function. Hence its evaluation function is:

$\boldsymbol{f(n) = c(n)}$ also referred to as $\boldsymbol{f(n) = g(n)}$

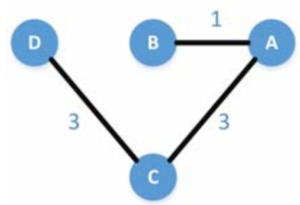
```
def UCS(problem):
    initial_state = problem.INITIAL_STATE()
    priority_queue = {}
    explored_set = {}
    priority_queue.enqueue(initial_state)

    # Continue until either a solution is found or all nodes explored.
    while( len(priority_queue) > 0):
        node = priority_queue.pop()
        # Must only check AFTER dequeueing the item to ensure it is optimal.
        if(problem.GOAL_TEST(node)): return SOLUTION(node)

        # Add the node to the explored set.
        explored_set.append(result)

        for action in problem.ACTIONS(node):
            result = problem.RESULT(node, action)
            # If not in the priority queue then enqueue it.
            if( result not in priority_queue and result not in explored_set):
                priority_queue.enqueue(result)
            # Current version of node has lower cost than version in priority queue
            elif( result in priority_queue and result.COST() < priority_queue[result]. COST()):
                priority_queue.remove(result)
                priority_queue.enqueue(result)
    # No path found
    return None
```

Pseudo code for A* and UCS is the same with the implementation of the COST() method.

A* Algorithm

A* algorithm is a combination of the benefits of **Greedy-Best First Search** and **Uniform Cost Search**.	**Evaluation Function $f(n)$:** $f(n) = g(n) + h(n)$ Also written as: $f(n) = c(n) + h(n)$	Only performs the **GOAL-TEST after the node has been dequeued** from the priority queue. Similar to Uniform Cost Search.	Derives from Dijkstra's Algorithm.

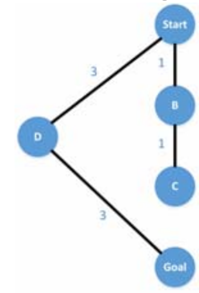
Example of A* Performing Better than Greedy Best First Search



Greedy Best First Search Oscillates Between Nodes A and B so it is Incomplete. This graph is solvable by A*.

Greedy Best First Search is **memory efficient** since it does not need to remember where it has been.

Example of DFS Performing Better than A*



Heuristic for A* is Euclidean distance. In this case, A* adds B then D to the frontier. It next expands B and adds C to the frontier. It next explores C and finds no solutions so it explores D then finds the goal.

Recursive Best-First Search

This algorithm is **optimal** when the heuristic is **admissible for trees**. The heuristic needs to be **consistent for tree search** to be optimal.

f_limit/min_eval_func_val – **Best alternative path available from the any ancestor of the current node.**

Simplified Description of Recursive Best First Search

1. Start from initial state and set the initial minimum cost of ∞

2. Generate all successors of current node. Set successor cost to either current node evaluation function value ($f(n)$) or the successors evaluation function cost.

3. Select successor node with minimum evaluation function ($f(n)$) cost.

4. If current node is a goal state, then return the solution.

5. If this cost is more than the current minimum, backtrack to find node with current minimum.

6. Extract the evaluation function cost ($f(n)$) of the second best successor of the current node.

7. Recurse using best successor found in step #3 and the minimum of the current minimum cost that was passed to the function and the second best successor of this node. This function results either a solution or None and updates the current best node's evaluation function cost ($f(n)$).

8. If step #7 returned a solution, then return that, otherwise, jump to step #3.

```
def RECURSIVE_DEPTH_FIRST_SEARCH(problem):
    return RDFS(problem, problem.INITIAL_STATE(), inf)

# Continues to recurse until current best cost is more than
def RBFS(problem, state, min_eval_func_val):
    # Check if a goal was reached. If so, return it.
    if( problem.GOAL_TEST(state) ):
        return SOLUTION(state)
    # Get set of successors
    for a in problem.ACTIONS(state):
        successors.append(problem.RESULT(state, a))
    # Check a successor exists
    If(len(successors) == 0):
        return None, ∞
    # Update all successor eval function values
    for s in successors:
        s.eval_func_val = max(node.eval_func_val, s.g + s.h)
    while(True):
        # Best successor is a node with min eval cost from successors
        best_successor = node with least eval function value from the successors
        # If the best successor is not better than current best, backtrack to current best
        if(best_succesor.eval_func_value > min_eval_func_value):
            return None, best_successor.eval_func_value
        # May need to recurse back to current level so store second best value for this level.
        second_best_successor_eval_func_val = Eval func value for second best successor of state
        # Run RBFS again from current node with the new min value the minimum of the current
        # minimum and the second best successor (i.e. alternative) for this current state/node.
        result, best_successor.eval_func_val = \
            RBFS(problem, best_successor, min(min_eval_func_val,
                                              second_best_succesor_eval_func_val)
        # If solution found, return it.
        if(result is not None):
            return result
```

Memory Bounded Heuristic Search

Iterative Deepening A* (IDA*) Algorithm

Variant of the A* algorithm that ***generally* slower but uses less memory.** Sets a maximum total cost (i.e. $f(n)$) to a starting value of μ. In each round, any node whose total cost (i.e. $f(n)$) is greater than the maximum is ignored. Perform A* for thresholds:

$$\mu < 2\mu < 3\mu < \cdots$$

```
def IDA_Star(problem, initial_max_cost, maximum_cost):
    current_max_cost = initial_max_cost
    while(current_max_cost < maximum_cost):
        result = A_Star_Search(problem, current_max_cost)
        if(problem.GOAL_TEST(result)):
            return result
        current_maximum_cost += initial_max_cost
    return None
```

Simplified Memory Bounded A*

Approach to save memory in A* algorithm.

Procedure:

1. Perform A* until you run out of memory.
2. Delete fringe or explored set node with the worst cost.

Heuristic Classification

Evaluation Functions $f(n)$ for Three Related Search Algorithms:

Uniform Cost Search: $f(n) = c(n)$	**Greedy Best First Search:** $f(n) = h(n)$	**A* Search Algorithm:** $f(n) = c(n) + h(n)$ A* algorithm is the only one of the three whose evaluation function estimates the cost of the **total solution.**

Admissible (Optimistic) Heuristic: Any heuristic that never over estimates the cost of a solution.	**Consistent (Monotonic) Heuristic:** For every node, *n*, every successor, *n'*, that is reached by action, *a*, then the cost to reach the goal from *n* is less than or equal to the actual cost to go from *n* to *n'* by action *a* ($c(n, a, n')$) plus the heuristic cost of *n'*. $h(n) \leq c(n, a, n') + h(n')$ **Note:** Any heuristic that is consistent is also admissible. **Example:** Triangle Inequality when the heuristic is straight-line distance.

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

Lemma #1 If $h(n)$ was a consistent heuristic, then the values of $f(n)$ are nondecreasing. **Given a node *n'* is a successor of *n* through action a, then:** $g(n') = g(n) + c(n, a, n')$ If *h(n)* is consistent, then: $h(n') + c(n, a, n') \geq h(n)$ Then: $g(n') + h(n') + c(n, a, n') \geq g(n) + h(n) + c(n, a, n')$ $f(n') + c(n, a, n') \geq f(n) + c(n, a, n')$ $f(n') \geq f(n)$	**Lemma #2:** Whenever A* selects a node for expansion, the optimal path to that node has been found. Had lemma #2 not been the case, then there would have been another node *n'* on the path from the start to *n* that would have been on the optimal path. **Because $f(n)$ is non-decreasing, this node would have had a lower value of $f(n)$ and would be expanded before *n* in A*. Hence, this is a contradiction.**	**Combining Lemma #1 and Lemma #2** **By Lemma #2:** If a goal node is explored, it is the optimal path to that goal node. **By Invariant of A*:** A* algorithm explores nodes in non-decreasing order of $f(n)$. **By Lemma #1:** $f(n)$ is nondecreasing. Combining Lemma #1, Lemma #2, and Invariant of A*: Paths to any other unexplored states, including goal states, will have evaluation function values ($f(n)$) greater than the first one explored. Hence, the optimal path to the first explored goal state is the optimal solution to the entire problem. Since by lemma #2 A* returns the optimal path to the first goal state, it returns the optimal path to the entire problem.

Choosing a Heuristic

Effective Branching Factor (b^*): For a set of *N* moves, it is the equivalent number of uniform branches for a depth *d*. It is a way to quantify the quality of a heuristic. $N + 1 = 1 + b + b^2 + \cdots + b^d$ $N + 1 = \frac{b^{*d+1} - 1}{b^* - 1}$ Derives from: $\sum_{i=m}^{n-1} a^i = \frac{a^m - a^n}{1 - a}$ **Best branch possible factor is 1.**	**Relaxed Problem:** A version of the actual problem with fewer restrictions. An exact solution to a relaxed problem is an admissible heuristic for the original problem.	**Dominating Heuristic:** A heuristic that always has a lower branching factor than another heuristic. **Composite Heuristic:** Given a set of **admissible** heuristics $\{h_1, h_2, \ldots, h_n\}$ none of which is dominating, then the best heuristic is the composite heuristic: $h_{composite} = \max\{h_1, h_2, \ldots, h_n\}$	

Subproblem: A reduced version of the actual problem. Admissible heuristics can be derived from the solution to subproblems.	**Pattern Database:** Stores the exact solution for all versions of a particular subproblem. To determine the heuristic cost for a version of the subproblem, look up the solution in the database and calculate the heuristic cost.	**Disjoint Patterns:** A problem can be divided into disjoint (i.e. nonoverlapping) subproblems. The disjoint solution to the problem is referred to as a disjoint pattern.	**Disjoint Pattern Database:** Stores solution to disjoint (non-overlapping, non-dependent) subproblems. Using multiple disjoint subproblems in a disjoint pattern database, you can come up with a composite heuristic by **summing** the cost to solve each individual subproblem.

Local Search

Local search generally operates using a single **current node** and generally moves to neighbors of that node.	If the local search problem is an **optimization problem**, then it is accompanied by an **objective function** that is to be maximized or minimized.	**Complete Algorithm:** Always finds a solution if it exists. **Optimal Algorithm:** Always finds a global maximum or minimum.	**State Space Landscape:** Landscape has a location (i.e. state) and an elevation (utility from the objective function)

Hill Climbing Algorithm

Local search algorithm that always proceeds to the next successor state with maximum utility. If two successors have the same utility, algorithm randomly chooses between them. Susceptible to **local maxima**.

Also referred to as **Greedy Local Search**.

Variants of Hill Climbing

Sideways Move: Allow hill climbing algorithm to move to a state of equal value. Helps to move past flat area in a graph. However, in a plateau, it can lead to an infinite loop so a limit on the number of consecutive sideways moves is common.

Stochastic Hill Climbing: Choose a successor state at random with the probability each successor is selected proportional to its utility.

Hill Climbing with Restarts: Hill climbing runs from a randomly chosen initial state. If it gets a solution, it returns. Otherwise, it generates another random initial state and repeats the process. Repeated *n* times or until a solution is found.

Example: If the probability of finding a solution from an initial state is p, then it is expected $\frac{1}{p}$ **restarts** will be required.

See page 122.

```
def HILL_CLIMBING_WITH_RESTART(problem, max_restarts):
    while( max_restarts > 0 ):
        max_restarts -= 1
        problem.INITIAL_STATE = problem.RANDOMIZE_STATE()
        result = Hill_Climbing(problem)
        if(problem.GOAL_TEST(result)):
            return result
    return None

def HILL_CLIMBING(problem):
    current_state = problem.INITIAL_STATE()

    while( True ):
        # Update the previous utility
        best_successor = None
        # Iterate through set of possible actions
        for action in state.ACTIONS():
            new_state = problem.RESULTS(state, action)
            if(best_successor is None
              or problem.UTILITY(new_state) > problem.UTILITY(current_state)):
                best_successor = new_state
        # Determine if the best successor is better than the current state
        if(problem.UTILITY(best_successor) > problem. UTILITY(current_state)):
            current_state = best_successor
        else:
            return current_state
    return None
```

Note: This is a goal based version of Hill Climbing. If you are simply searching for a maximum or minimum, you would need to modify the algorithm to return "current_state" at the end.

Simulated Annealing

Can be used for either maximization or minimization problems.

Algorithm is designed to allow the current_node to move to a worse state with decreasing probability as time progresses.

Probability of Moving to a Lower Value Solution is:

$$P = e^{-\frac{\Delta k}{schedule(t)}}$$

Simulated annealing chooses **a random successor.**

```
import math
import random
def SIMULATED_ANNEALING(problem, schedule, limit, t_min):
    current_state = problem.INITIAL_STATE()
    t = 0
    while( True ):
        t += 1
        T = schedule(T)
        if(T < t_min or problem.GOAL_TEST(current_state)):
            return current_state
        # Get the set of actions.
        actions = current_state.ACTIONS()
        # If no successors possible, terminate
        if(len(actions) == 0):
            return None
        # Randomly select a successor
        a = actions[random.randint(0, len(actions) – 1]
        # Get the successor state
        next_state = problem.RESULT(current_state, a)
        # Calculate the error
        error = problem.UTILITY(next_state) - problem.UTILITY(current_state)
        # If error is positive or probability less than specified number, then update the current state.
        if(error > 0 or random.random() < math.exp( error/ T ):
            current_state = next_state
```

Note: This version of the code is a maximization problem. Would need to modify slightly for a minimization problem.

Local Beam Search

Type of local search.

Procedure:

1. Begin with *k* randomly generated states.
2. Check if any descendent states at the goal. If so, return state.
3. Order all successors from the *k* states and sort them by decreasing performance.
4. Choose the best *k* successors. If any successor has performance measure better than the current best, return to step #2.

The *k* successors are considered a **pool of candidates**. The successors are considered **offspring**.

Variant of Local Beam Search

Stochastic Local Beam Search: Choose *k* successors stochastically based off some metric.

Genetic Algorithm

A genetic algorithm is a **stochastic beam search** algorithm with one key modification: • In local beam search, successors come from **modifying a single state (asexual reproduction)**. • In genetic algorithm, successors come from **combing two parent states (sexual reproduction)**. **Population**: Set of *k* solutions. The **initial population** is *k* randomly generated solutions. **Individual:** One solution/state in the population. **Fitness Function:** Evaluation function that rates the quality (i.e. fitness of a solution) generally with general condition that better states have higher fitness function value. **Crossover:** Process of merging two solution states to form a new successor. **Mutation:** Random change to a successor solution.	See code below

```
def GENETIC_ALGORITHM(problem, FITNESS_FUNCTION, t_max)
    #  Generate the population.
    population = problem.GENERATE_POPULATION()
    # Start at time 0.
    t = 0
    while(t < t_max or Not problem.GOAL_TEST(best_solution)):
        # Increment current time.
        t += 1
        new_population = {}
        best_solution = None
        for i in range(0, problem.POPULATION_SIZE()):
            # Select two parent solutions.
            x = RANDOM_SELECTION(population, FITNESS_FUNCTION)
            y = RANDOM_SELECTION(population, FITNESS_FUNCTION)
            # Merge the two solutions
            child = REPRODUCE(x, y)
            # Mutate on a low probability
            if(random.random() < problem.MUTATION_PROBABILITY):
                problem.MUTATE(child)
            if(best_solution is None or problem. UTILTY(best_solution) < problem.UTILTY(child)):
                best_solution = child
            # Add the child solution to the new population.
            new_population.append(child)
        # Set the population to the newly created set.
        population = new_population
    return best_solution

def REPRODUCE(x, y):
    # Pick a random cross over point
    crossover_point = random.randint(0, len(x) – 1)
    # Crossover the two halves
    return x[0:crossover_point] + y[crossover_point:len(y)]
```

8-Puzzle Goal State:

X	1	2
3	4	5
6	7	8

Minimax (Adversarial Search)

Adversarial search problems are those search problems that arise in **multiagent**, **competitive** environments. Adversarial search problems are also known as **games**.

In a **zero-sum game**, the results for the two players are always **equal and opposite**.

Optimal Strategy – A sequence of contingent decisions that will lead to outcomes as least as good as any other sequence of decisions against an infallible player.	**Perfect Information** – Any situation where an agent has all relevant information with which to make a decision and the results of actions are **deterministic**.	**Minimax Value** – Utility of being in a current state assuming both players play optimally until the end of the game.

$$H-MINIMAX(s,d)=\begin{cases} UTILITY(p), & \text{if } \mathbf{CUTOFF\_TEST}(s,d) \\ \max_{a\in ACTIONS(s)} H-MINIMAX(RESULT(s,a),d), & \text{if } \mathbf{PLAYER}(s) \text{ is MAX} \\ \min_{a\in ACTIONS(s)} H-MINIMAX(RESULT(s,a),d), & \text{if } \mathbf{PLAYER}(s) \text{ is MIN} \end{cases}$$

Initial State in Minimax – s_0

Given a state, *s*, the six key methods used on that state are:

1. PLAYER(s) – Returns active player for the current state
2. ACTIONS() – Set of all possible actions/moves that can be made.
3. RESULTS(s,a) – Given a state, *s*, and an action *a*, it returns the successor state. It is also called a **Transitional Model**.
4. CUTOFF_TEST(s,d) – Used in Heuristic minimax. Given a state, s, and a recursive depth, d, it determines if the cutoff condition of either a maximum depth or goal state has been reached.
5. TERMINAL_TEST(s) – Used in standard minimax. Given a state, *s*, this function returns whether a goal state has been met. **Terminal states** are **leaf nodes** in the **search tree.**
6. UTILITY(s) – Given a state, *s*, this function returns the state's utility score. It is also called a **Utility Function.**

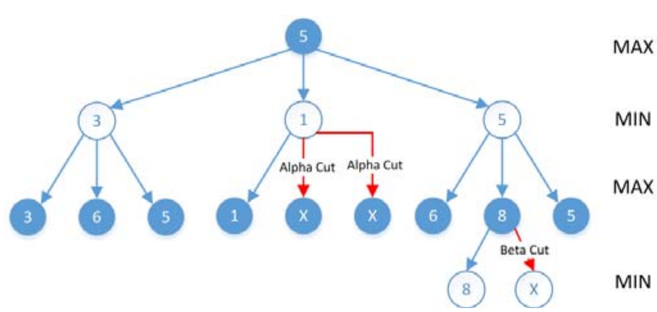
Time Complexity with Alpha-Beta Pruning: $O\left(b^{\frac{d}{2}}\right)$	**Time Complexity without Alpha-Beta Pruning:** $O(b^d)$

```
def Minimax_Algorithm(state, is_max):
    alpha_max = -inf
    beta_min = inf
    best_successor = None
    # Iterate through all possible actions from this state
    for a in state.ACTIONS():
        # Get the successor state
        next_state = state.RESULT(state,a)
        # Call heuristic minimax with starting depth 0
        score = H-Minimax(next_state, 0, !is_max,
                          alpha_max, beta_min)
        if(is_max and score > alpha_max):
            best_successor = a
            alpha_max = score
        elif(not is_max and score < beta_min):
            best_successor = a
            beta_min = score
    # Return the move with the best score
    return best_move
```

```
def H-Minimax(state, depth, is_max, alpha_max, beta_min)
    # p is the reference player for the utility function. Typically max.
    if ( state.CUTOFF-TEST(depth) ):
        return state.UTILITY(p)
    for a in state.ACTIONS():
        next_state = state.RESULT(state, a)
        if(is_max):
            # Perform beta pruning
            alpha_max = max(alpha_max, H-Minimax(next_state, depth+1,
                                 not is_max, alpha_max, beta_min))
            if(alpha_max ≥ beta_min):
                return alpha_max
        else:
            beta_min = min (beta_min, H-Minimax(next_state, depth+1,
                                not is_max, alpha_max, beta_min))
            # Perform alpha pruning
            if(alpha_max ≥ beta_min):
                return beta_min
    # After all actions tested, return score.
    if(is_max):
        return alpha_max
    else:
        return beta_min
```

Alpha Beta Pruning

Alpha (α) – Maximum value found along the path by the MAX player. **Alpha Cut/Alpha Pruning** – Performed by the **MIN player**. When the MIN player's minimum score is already less than a previous MAX player's maximum score, stop investigating subsequent paths and return the **current minimum score**.	**Beta (β)** – Minimum value found along the path by the MIN player. **Beta Cut/Beta Pruning** – Performed by the **MAX player**. When the MAX player's maximum score is greater than a previous MIN player's minimum score, stop investigating subsequent paths and return the **current maximum score**.



Minimax Search Tree Example with Alpha and Beta Cuts.

This is a three move/**ply** search tree.

Constraint Satisfaction Problem

Search problems deal with states that are **atomic** (i.e. indivisible).

Often a state has field variables. Such field values are called a **factored representation** of the problem. A state **solves** a factored representation if each field variable satisfies all constraints on that variable.

A factored representation can allow you to eliminate large areas of the search space by identifying then ignoring variable/value combinations that violate constraints.

A constraint satisfaction problem **solution** is an assignment of values to variables that satisfies all constraints.

Assignment of values to variables in CSPs is **commutative**. Hence, the order that the values are assigned do not matter. If you consider the problem a search tree, there are at most *d* children from each node leaving a total of d^n solutions for a finite domain

Components of a Constraint Satisfaction Problem: 1. ***X* – Set of variables** $\{X_1, X_2, X_3, \dots, X_n\}$ 2. ***D* – Set of Domains** $\{D_1, D_2, D_3, \dots, D_n\}$ 3. ***C* – Set of Constraints** $\{C_1, C_2, C_3, \dots, C_n\}$ **Optional Definition:** R - Relation of multiple variables $R(X_i, \dots, X_m)$	**Definition of a Constraint** A constraint is a pair: $< scope, relation >$ **Scope:** Tuple of variables that participate in the constraint **Relation:** A relation that the variables can take on.

Assignment – Allocation of values to variables. **Solution:** A complete and consistent assignment.	**Consistent Assignment** – An assignment of values that does not violate any constraints. This leads to the term **consistency** which is the **satisfaction of constraints.**	**Complete Assignment** – Every variable is assigned a value.	**Partial Assignment** – Only a subset of variables are assigned a value.

Domain A variable's domain can be either **discrete** or **continuous**. If it is discrete, it can be either **finite** or **infinite** (e.g. set of integers). **Simplest CSP Type:** Finite, discrete domain	**Constraint Language** Defines the allowed relations between variables. It eliminates the need to enumerate allowed value lists.	**Linear Programming Problem:** Continuous CSP with **linear constraint function**(s). Constraint functions can also be **nonlinear**.

Constraint Types

$X = \{X_1, X_2\}$ and $D = \{A, B\}$ **Example Constraint:** $C = \langle (X_1, X_2), rel \rangle$ with $rel = \{(A, B), \quad (B, A)\}$	**Precedence Constraint:** A constraint that forces one variable to occur before (i.e. be less than) another variable. **Example:** $T_1 + d \leq T_2$	**Disjunctive Constraint:** A constraint that two variables do not overlap (i.e. are not equal): **Example:** $T_1 + d \leq T_2$ or $T_2 + d \leq T_1$	**Absolute Constraint:** Any constraint that must be met.	**Preference Constraint:** A constraint which guides the solution to preferred values. Problems that optimized preference constraints are called **constraint optimization problems**.

Unary Constraint – A constrain involving only a single variable.	**Binary Constraint** – A constrain involving exactly two variables.	**Higher Order Constraint:** A constraint that involves a **fixed** number of variables that is more than two. **All higher order constrains can be reformed as a set of binary constraints.**	**Global Constraint:** A constraint that takes an **arbitrary** number of variables. It does not need to be all variables. It just needs to be **not fixed** (i.e. arbitrary). **Example:** ***Alldiff***

Constraint Graph/CSP Network: Representation of a CSP as a graph. Each node is a variable and the arcs are binary constraints.

Inference: Using known/assigned values for a set of variables to select the values for other variables.

Constraint Propagation: Using the constraints to reduce the number of legal values for a variable. This in turn reduces the number of legal values for other variables in a cycle.

Local Consistency: Given a constraint graph, enforcing consistency (i.e. ensuring variables satisfy constraints) locally **in each part of the graph** leads to invalid values being eliminated throughout the graph.

Node Consistency

Node Consistent Variable – Any variable where every value in the variable's domain **satisfies all of its unary constraints** in a CSP network.	**Node Consistent Network** – Any CSP network where **all variables are node consistent**.	Node consistency can be done as a **preprocessing step** to eliminate invalid values.

Arc Consistency

Arc Consistent Variable – Any variable where every value in the variable's domain **satisfies all of its binary constraints** in a CSP network.

Variables are arc-consistent with respect to one another. Example: X being arc consistent with respect to Y does **NOT** imply Y is arc consistent with respect to X.

Arc Consistent Network – Any CSP network where **all variables are arc consistent**.

AC-3 (Arc Consistency Algorithm #3)

Algorithm used to solve for Arc consistency
Only possible with finite domains.

Constraints in Arc Consistency Algorithm

In each iteration of AC-3 algorithm, it only checks the variable being arc-constrained (example in constraint (X,Y), X is being constrained by Y). To have a two directional constraint for X and Y, arc queue would need to contain (X, Y) and (Y, X)

After reducing the domain of X from constraint (X, Y), algorithm needs to recheck any domains that were constrained by X to ensure its domain values are still valid.

Running Time of AC-3 Algorithm

1. **REVISE Function**: $O(d^2)$
For each value in the domain of X_i (up to *d* elements), you iterate overall elements in the domain of X_j. Hence the running time is:

$$O(d*d) = O(d^2)$$

2. **Number of Times REVISE function is Run Per Constraint:** $O(d)$
The REVISE function is run whenever a constraint is popped off the queue. If the domain size is queue, it can be popped off the queue up to *d* times (once for each element.

3. **Number of Constraints:** *c*

Total Running Time:

$$O(c) \cdot O(d) \cdot O(d^2) = O(cd^3)$$

```
def AC_3(csp):

    arc_queue = []
    # Add all binary constraints to the queue.
    for b_constraint in csp.BINARY_CONSTRAINTS:
        arc_queue.append( (b_constraint.X_i, b_constraint.X_j )

    # Iterate until all arcs have been made consistent or an inconsistency is found.
    while( len(arc_queue) > 0 ):
        (X_i, X_j) = arc_queue.pop()
        # Check if the domain of X_i is revised.
        if( REVISE(csp, X_i, X_j) ):
            if(len(X_i) == 0 ):
                return False
            # Only X_i's domain is reduced in function "REVISE" so only check relative to that.
            # Since X_i's domain is reduced, any variable that is constrained by X_i may need to be reduced
            for X_k in X_i.NEIGHBORS() – {X_j}:
                # Only add back to domain if not X_j
                if(X_k != X_j and (X_k, X_i) not in arc_queue):
                    arc_queue.append( (X_k, X_i) )
    return True

def REVISE(csp, X_i, X_j):
    revised = False # Confirmed in loop
    # Verify all elements in the domain of X_i have a corresponding value in X_j.
    for x in csp.D_i:
        constraining_value_exists = False
        # Iterate through all elements in X_j's domain to see if it constrains x in X_i.
        for y in csp.D_j:
            if( (x,y) in csp.C(X_i, X_j)) :
                constraining_value_exists = True
                break
        # If no constraining value exists in X_j, then remove the value from X_i.
        if(not constraining_value_exists):
            csp.D_i.remove(d)
            revised = True
    # Return whether the domain of X_i was revised (i.e. reduced)
    return revised
```

Page 209

Path Consistency

Path Consistency – **A two variable set (X_i, X_j) are path consistent with respect to a third variable X_m** if for every assignment of values to X_i and X_j consistent with the constraint $\{X_i, X_j\}$, there is a valid assignment to X_m that satisfies the constraints $\{X_i, X_m\}$ and $\{X_m, X_j\}$.

Origin of the Term "Path Consistency"

Given a two variable set $\{X_i, X_j\}$ that is path consistent with respect to a variable X_m, then it is like X_m is on the path between X_i and X_j.

Algorithm to Solve to Check for Path Consistency: PC-2

k-Consistency

A CSP is ***k*-consistent** if for **any set of *k-1* variables** and for **any consistent assignment** to those variables, a consistent value can **always** be assigned to any *k*-th variable.

Proving *k*-consistency takes exponential and space in the worst case.

1-consistency is **node consistency**.

2-consistency is **arc consistency**.

Strongly *k*-consistent: Any CSP that is 1-consistent and 2-consistent and 3-consistent through k-consistent. Hence it is consistent for variable sets of size 1 through *k*.

Given *n* variables and a CSP that is strongly *n*-consistent, then an assignment of values is possible for this CSP.

Running Time to Solve *n*-Consistent CSP

Time Complexity: $O(n^2d)$

Running time derives since for every *i*-th variable to assign, you must check all *i-1* variables for every *d* elements in the domain. Hence:

$$d \cdot \sum_{i=1}^{n} i - 1 = d \cdot \left(\frac{n \cdot (n+1)}{2} - n\right) = O(dn^2)$$

Consistency Checks for Global Constraints

Global Constraint – A constraint with an arbitrary number of variables.

Example Global Constraint: *Alldiff*

Alldiff Consistency Algorithm

1. Delete a variable that has a singleton domain.
2. Remove the value from the domains of all other variables.
3. If any singleton domain variables still exists, jump to step #1.
4. If a domain has no values or there are more values than there are variables, the *Alldiff* constraint fails.

Simplified Explanation of Alldiff Consistency Check

If there are *m* variables and *n* possible values and $m > n$, then an inconsistency exists.

Sudoku

Square grid of n by n cells. All numbers in a row must be unique and all numbers in a column must be unique. For every $\sqrt{n}$ by $\sqrt{n}$ subgrid, all numbers must be unique. Each section of the board where all numbers must be unique (e.g. row, column, subgrid) is called a **unit**.

Formal Definition of Sudoku as a CSP:
Variables: n^2 total variables (one for each cell).
Domain: $\{1, 2, 3, \ldots, n\}$
Constraints: $3n$ *Alldiff* constraints for each unit.

AC-3 Algorithm can be used to infer the value of cells and to reduce the domains of cells.

CSPs and Backtracking

Backtracking Search – Variant of Depth First Search where values are assigned to variables until no consistent, legal assignments are possible for a given variable at which point the algorithm ***backtracks*** to try to reassign a previous variable to a new value.

Key Functions in Backtracking Search
1. **SELECT_UNASSIGNED_VARIABLE**
2. **ORDER_DOMAIN_VALUES**
3. **INFERENCE**
4. **BACKTRACK** (recursion)

See page 215.

```
def BACKTRACKING_SEARCH(csp):
    return BACKTRACK({}, csp)

def BACKTRACK(assignment, CSP):
    # Consistency of all variable assignment checked so if assignment is complete, it is a solution.
    if(csp.COMPLETE_ASSIGNMENT(assignment)) return assignment

    # Select the next variable to assign
    next_var = csp.SELECT_UNASSIGNED_VARIABLE()

    # Order the domain values based off which want to check first
    var_doman = csp.ORDER_DOMAIN_VARIABLES(assignment, next_var)

    # Iterate through all domain values.
    for d in var_domain:

        # Ensure the assignment is consistent.
        if(csp.CONSISTENT_ASSIGNMENT(assignment, d)):
            # Add the variable value to the assignment
            assignment[var_domain] = d

            # Get and apply any inferences
            inferences = csp.INFERENCE(assignment)
            # Only recurse if valid inferences found.
            if(inferences is not None):
                assignment.APPLY_INFERENCES(inference)
                result = BACKTRACK(assignment, csp)
                if( result is not None):
                    return result
                assignment.REMOVE_INFERENCES(inference)
            # Since no solution found using this assignment and variable value
            # remove this variable value from the assignment.
            remove( assignment[var_domain] )

    # No solution found so return None for failure.
    return None
```

Making Backtracking Search More Efficient and Sophisticated

Variable Ordering

By selecting a **variable most likely to fail earliest,** you are **prune the search tree** and **reduce the effective branching factor**.

Minimum Remaining Value (MRV), Fail First, Most Constrained Variable Heuristic: Select the variable to assign next that has the smallest inferred domain (i.e. least remaining legal values).

Degree Heuristic: Select the variable for expansion that has the largest number of constraints on other variables. **Most commonly used heuristic to select the first variable for assignment.**

Degree heuristic can be used as a **tie breaker** for the more powerful MRV heuristic.

Value Ordering

Least-Constraining Value Heuristic: Select the value that rules out the least number of values for neighboring variables in the graph.

Interleaving Search and Inference

AC-3 can be used to infer reductions in the search domain both **before and during search**.

Forward Checking – One way to implement "Inference" in Backtracking algorithm. Whenever a variable is assigned, establish arc consistency for it on all unassigned variables. If arc consistency checking was done in preprocessing, forward checking adds no value.

MRV can be combined with forward checking to further prune the search tree.

Chronological Backtracking: Simplest form of backtracking. **Revisit the last assigned variable** (i.e. **most recent decision**) before the current variable. If the previous variable does not constrain the current variable, backtracking to only that level is wasteful.

Intelligent Backtracking

Better to backtrack to a variable that may fix the consistency issue.

Conflict Set: Set of value assignments that conflict with a some value for a variable. **Note:** This is value assignments not variables since a variable that can conflict for one value does not conflict for the currently assigned value.

Backjumping: Backtracking to the most recent variable in the conflict set.

Variable ordering is fail-first ordering while **value ordering is fail-last**. This is because when you are trying to fail-first by selecting a variable, the order you inspect the values does not matter as you need to **inspect them all anyway**. As such, it makes the most sense to inspect the best solutions first in case one of them ***does actually succeed***.

Logical and Knowledge Based Agents

Knowledge Base (KB) – Central component of a knowledge based agent. Composed of a set of **sentences**. **Similar to a database.**

Knowledge Representation Language – Formal notation used to express sentences in the knowledge base (KB).

Sentence – Statements that define the knowledge based. They have a specific notation called a syntax and their value (i.e. true or false) is defined by the semantics.

Axiom – A sentence that is taken as given without being derived from other sentences.

Inference – Deriving new sentences from existing sentences.

Valid Knowledge Base Operations:

1. **TELL**
2. **ASK**

Supporting Knowledge Based Agent Commands:

1. **MAKE_PERCEPT_SENTENCE**
2. **MAKE_ACTION_QUERY**
3. **MAKE_ACTION_SENTENCE**

Background Knowledge – Initial knowledge in the knowledge base.

Four Step Procedure for a Knowledge Based Agent:

1. **Tell the knowledge base what it perceives.**
2. **Ask the knowledge base it should perform.**
3. **Tell the knowledge base the action it will perform.**
4. **Executive the action.**

```
def KNOWLEDGE_BASED_AGENT()
    # KB is the persistent knowledge base.
    # t a time counter initially starting at 0.

    TELL( KB, MAKE_PERCEPT_SENTENCE(t) )
    action = ASK(KB, MAKE_ACTION_QUERY(t) )
    TELL ( KB, MAKE_ACTION_SENTECE(t) )

    t += 1 # Increment time

    # Return the selected action.
    return action
```

Knowledge Level – What the agent knows at a give point in time. Given an agent's knowledge level and goals, you can predict its actions.

Declarative Approach – Tell the knowledge base all it needs to know.

Procedural Approach – Procedures for desired behaviors and actions are hard coded into the agent.

Wumpus World

	Performance Measure	Actuators	Sensors
The knowledge based agent is in an environment consisting of rooms connected by passageways. Some rooms contain bottomless pits while others contain goal. One wumpus lives in the cave in one room. Wumpus eats anyone who enters its room but does not move. Player has one arrow that can kill the wumpus.	+1000 points for getting gold. -1000 points for falling into a pit or eating a wumpus. -1 for each action taken. -10 for using an arrow.	Move forward one room. Turn left 90 degrees. Turn right 90 degrees. Shoot the arrow Climb out (if in starting space)	**Stench:** A wumpus is in an adjacent room. **Breeze:** A pit is in an adjacent room. **Glitter:** Gold is in the player's room **Scream:** Wumpus is killed. **Bump:** Player walks into a wall.

Logic

Syntax – Sentence formatting to make all knowledge sentences well formed.

Semantics – Provide meaning to sentences. It defines **truth** for every **possible world**.

Example: For the sentence, $x + y = 4$ is true in the world where $x = 2$ and $y = 2$.

Model – Substitute for the phrase "**possible world.**" **A model fixes the truth or falsehood for every relevant sentence.**

Satisfaction: Making a sentence true using an allowed model/possible world.

Example: If sentence α is true in model m, then model m **satisfies** sentence α.

Entailment

Entailment Between Sentences: When one sentence logically follows from another sentence or set of sentences. It is similar to implies in philosophy.

Symbol: ⊨

Given two sentences α and β, then sentence α entails the sentence β if and only if:

$$\alpha \vDash \beta \Leftrightarrow \forall M(M(\alpha) \subseteq M(\beta))$$

The knowledge base is a set of sentences. The knowledge base is false in models that conflict with the knowledge base.

Model Checking: Given a knowledge base, KB, and verify it is a model of α. Hence:

$$M(KB) \subseteq M(\alpha)$$

Model checking entails enumerating all possible models to determine whenever *KB* is true that α is also true. It only works on finite domains.

Logical Inference: Process of drawing conclusions (i.e. new sentences) through entailment.

Symbol of Inference: ⊢

Given a knowledge base, *KB*, and a sentence α, if an inference algorithm, *i*, inferred α from *KB* then:

$$KB \vdash_i \alpha$$

Sound or Truth Preserving Inference Algorithm: Can only derive entailed sentences. **Hence it cannot prove any sentence that is wrong.**

Example: Model checking is a sound algorithm since it does not work on infinite spaces.

Complete Inference Algorithm: Can derive any entailed sentence. **A complete inference algorithm can prove anything that is right.**

Syntax

Syntax: Defines allowable sentences.
Semantics: Defines what a sentence means.
Model: Fixes the **truth value** (i.e. true or false) for each proposition symbol.

Atomic Sentence: Simplest type of sentence and contains a single **propositional symbol** (i.e. **variable**)

Propositional Symbol: Represents a proposition or statement that can be either true of false.
Naming Convention: First letter is capitalized followed by lower case letters and subscripts.
Positional symbols with fixed meaning: True (always true position) and False (always false proposition)

Logical Connectives

Symbols that operate on propositional logic symbols.

¬: Not (**Negation**)
∨: Or (**Disjunction**). Individual terms are called **disjuncts**.
∧: And (**Conjunction**). Individual terms are called **conjuncts**.
⇒: Imply (**Implication**)
⇔ or ≡: **Biconditional.** "**If and only if**"

***A* ⇒ *B* is True unless *A* is true and *B* is false. *A* ⇔ *B* is true only if A and B are both true or are both false.**

If $A \Rightarrow B$, then:

- A is the **premise** or **antecedent**
- B is the **conclusion** or **consequent**.

Valid Sentence

$$AtomicSentence \coloneqq True|False|P|Q|R$$
$$Sentence \coloneqq AtomicSentence|Sentence$$
$$ComplexSentence \coloneqq (Sentence)|\,[Sentence]$$
$$|\neg Sentence$$
$$|\,Sentence \lor Sentence$$
$$|\,Sentence \land Sentence$$
$$|\,Sentence \Rightarrow Sentence$$
$$|\,Sentence \Leftrightarrow Sentence$$

Operator Precedence

$$\neg,\ \lor,\ \land,\ \Rightarrow,\ \Leftrightarrow$$

Inference Proving

Checking if $KB \vDash \alpha$

Model Checking: Enumerate all the models and check if all for all possible models where KB is that α is also true. **Model checking is very similar to a truth table.**	**Theorem Proving:** Using sentences already in the model, apply rules of inference to construct a proof of the desired sentence without consulting models.

Literal: In a complex sentence, a literal is either an atomic sentence (i.e. **positive literal**) or its negation (i.e. **negative literal**).	**Logical Connectives:** Used to construct complex sentences out of atomic sentences.

Logical Equivalence: Two sentences α and β that are true in the same set of models. **Notation:** $\alpha \equiv \beta$	**Validity**: A sentence that is **valid** (**true**) in **all models**. **Tautology**: A valid sentence.

Common Logical Equivalences

Commutative of ∧	$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha)$	**Commutative of ∨**	$(\alpha \vee \beta) \equiv (\beta \vee \alpha)$
Associativity of ∧	$\left((\alpha \wedge \beta) \wedge \gamma\right) \equiv \left(\alpha \wedge (\beta \wedge \gamma)\right)$	**Associativity of ∨**	$\left((\alpha \vee \beta) \vee \gamma\right) \equiv \left(\alpha \vee (\beta \vee \gamma)\right)$
Double Negation	$\neg(\neg \alpha) \equiv \alpha$	**Contraposition**	$(\alpha \Rightarrow \beta) \equiv (\neg \beta \Rightarrow \neg \alpha)$
Implication Elimination	$(\alpha \Rightarrow \beta) \equiv \neg \alpha \vee \beta$	**Biconditional Elimination**	$(\alpha \Leftrightarrow \beta) \equiv \left((\alpha \wedge \beta) \vee (\neg \alpha \wedge \neg \beta)\right)$
DeMorgan's Law	$\neg(\alpha \wedge \beta) \equiv (\neg \alpha \vee \neg \beta)$	**DeMorgan's Law**	$\neg(\alpha \vee \beta) \equiv (\neg \alpha \wedge \neg \beta)$
Distributivity of ∧ and ∨	$(\alpha \wedge (\beta \vee \gamma)) \equiv \left((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)\right)$	**Distributivity of ∧ and ∨**	$(\alpha \vee (\beta \wedge \gamma)) \equiv \left((\alpha \vee \beta) \wedge (\alpha \vee \gamma)\right)$
Modus Ponens	$(\alpha, \alpha \Rightarrow \beta) \equiv \beta$	**Modus Tollens**	$(\neg \beta, \alpha \Rightarrow \beta) \equiv \neg \alpha$
And Elimination	$(\alpha \wedge \beta) \Rightarrow \alpha$		

Satisfiability: A sentence that can be made true with some model. For a finite environment, satisfiability can be by enumerating all possible models and seeing if any leads to the statement being true. CSP consistency checking is a type of satisfiability problem. **Example:** $(\alpha \wedge \beta)$ is true in the model: $M = \{\alpha = True, \beta = True\}$	**Validity and Satisfiability:** A sentence is valid if and only if its negation is not satisfiable. **Reduction ad absurdum/Proof By Reduction/Proof by Contradiction**: Given a logical expression, assume the opposite of the expression and determine if it is satisfiable.

Proof: A chain of conclusions that leads to the establishing some statement following from the knowledge base.

Example

Consider a situation where four light switches on a control panel. Define a knowledge base for this system with conditions defined in **Part A** and **Part B**. **Definition:** S_1: Propositional symbol for the first switch and is true if the switch is on and false otherwise. S_2: Propositional symbol for the second switch and is true if the switch is on and false otherwise. S_3: Propositional symbol for the third switch and is true if the switch is on and false otherwise. S_4: Propositional symbol for the fourth (i.e. last) switch and is true if the switch is on and false otherwise.	**Part A:** The first and last switches are never both on. $\neg(S_1 \wedge S_4)$ $\neg S_1 \vee \neg S_4$ **Part B:** At least one switch must be on. $S_1 \vee S_2 \vee S_3 \vee S_4$

Python Review

Python Basics

Command Line Call to Run Python:
python filename.py
Python File Extension:
*.py

Command to Print to Console:
print "Hello World!"
Printing without Inserting a Newline:
Use "," (Comma)

print "Hello World"**,**

Command to Get Last Result:
_ (Underscore)
Example:
>>> 2/3 + 7.9
>>> print _ + 1 # prints 8.9

Valid Python Operators:
+, *, -, /, *=, /=, -=, +=, %, ==, !=
// (Integer Division), ** (Power)

Math Functions:
math.exp(*value*): e^value
random.randint(n,m): Integer n ≤ x ≤ m
random.random(): Float 0 ≤ x < 1

Invalid Operators:
++, --

Minimum and Maximum Value:
inf, -inf

Conditionals:
if(*expr*):
  # Do something
elif(*expr*):
  # Do something
else:
  # Do something

Boolean Arithmetic:
is, and, or, not

Boolean Literals:
True, False

Check Membership in List:
in

File IO:
f = *open*("filename.txt", "w")
line = f.*readline*()
f.*close*()
# Iterate over a file line by line
for line in open("my_file.txt"):
  #Do something

Formatted Printing:
Use the % symbol similar to C/C++

print "%3d %0.2f" % (10, .9799)
Prints "10 0.98"

Python String Manipulation

Python String Implementation
Immutable ***list*** of characters.

String Concatenation:
+ (plus sign)

Converting from a String:
- **int**("38")
- **float**("46.456")

Converting to a String:
- **str**(7)
- **repr**(32.9)

Substring Manipulation
Use [] like a list with the first character index 0
a = "Hello World"
print a[4] # Prints "o"
print a[:5] # Prints "Hello"
print a[6:] # Prints "World"
print a[3:8] # Prints "lo Wo"

Checking for Substring:
Use the *in* operator:
if("hello" in "hello world"):
  print "It's in there."

Get Index of Substring:
x = "hello world".index("llo")
print x # Prints "2"

Element Containers

List (Array) Basics:
Able to hold data of different types in the same list including other lists. Uses **[]**
x =[5, 4, "hello", "world"]
print x[1] # Prints "4"
print x[1:] # Prints "[4, "hello", "world"]"
print x[0:2] # Prints "[4, 5]"
y =[[3, 2], [1, 0]]
print y[1][0] # Prints 1

Nested (Two-Dimensional) Lists:
y =[[3, 2], [1, 0]]
print y[1][0] # Prints 1
Concatenating Lists:
x = [1, 2, 3]
y = [4, 5]
z = x + y
print z # Prints "[1, 2, 3, 4, 5]"

List Length:
Use **len**()
x = [1, 2, 5, 10]
print len(x) # Prints "4"
Extracting List Properties:
max(*list*) – Gets Maximum Value in List
min(*list*) – Gets Maximum Value in List

Tuple:
Immutable list. Created used **()** parenthesis.
Accessing Tuple Elements:
c = (4, 5)
print c[1] # Prints "5"
a, b = c # a = 4 and b = 5

Creating a Tuple:
a = (1, 2, 3) # Tuple of size 3
b = (x, y) # Tuple made of two variables
c = "Hello", "World" # Tuple of size 2
d = () # Empty Tuple
e = "yo", # Tuple of size 1
f = ("yo",) # Equal to e
g = (d,) # Tuple of empty tuple ((),)

Sets:
Unordered collection of unique elements.
x = set([3, 6, 9, 2])
my_set = set("goodness")
print my_set # Prints ["g", "o", "d", "n", "e", "s"] with **no duplicates**
Frozenset:
An immutable set.
x = frozenset([4, 5, 6])
Set Operations:
| Union, & Intersection, - Difference,
^ Symmetric Difference (XOR)

Dictionary:
Associative Array (i.e. hash table). Uses **{}** curly brackets.
person ={
  "name": "bob",
  "age": "27",
  "sex": "Male"
}

print person["name"] # prints "Bob"
Deleting from a Dictionary:
del person["name"]

Dictionary Membership Test:
Use the keyword "in"
if("name" in person):
  print person["name"] # Prints "bob"

Accessing Tuple Elements:
person.**keys**() # Gets all dict keys
person.**values**() # Gets all dict values
person.**len**() # Gets all dict length

Looping and Iteration

While Loop:
while(*expr*):
  # Do something

For Loop:
for x in [2, 4, 5, 6, 9]:
  print x
for y in range(1, 10):
  print y # Only prints 9 lines

range:
Iterable object in Python.
range(0, 10) – Creates list of 0 to 9 in steps of 1
range(10) – Starting 0 not needed. Same as range(0,10)
range(0, 5, 2) – Starts 0 and steps by 2 until 5
range(7, 2, -1) – Starts at 7 and decrements by 1 until 3

range vs. xrange:
range creates an array that Python iterates over. This is memory inefficient. *xrange* acts like a real for loop without the memory overhead of range.

Iterable Objects in Python:
set, frozenset
List, Tuple
Dictionary *key*
File (*open*("filename")
String (letter by letter)
Generator

Functions

Creating a Function:
Keyword: ***def***

```
def my_func(params):
    # Do something
```

Keyword to Return: return

Supports Recursion: Yes

Taking an Arbitrary Number of Input Variables
Keyword: *args

```
def my_function(*args):
    pass
```

Scope:
Default scope in python is **local**.

```
i = 5
def print_i():
  i = 4
  print i
print_i() # Prints "4"
print I # Prints "5"
```

Keyword to Add to Global Scope:
global

```
def assign_i():
  global i
  i = 3
```

Storing a Function in a Variable:

```
def print_i()
  i = 4
  print i

a = print_i
a() # Prints "4"
```

Anonymous Function:
Keyword: **lambda**

```
g = lambda x: x**3
print g(10)  # Prints "1000"

h = lambda y,z: z + 2*y
print h(2, 3)  # Prints "8"

def make_adder(n):
    return lambda z: z+n

f = make_adder(2)
print f(3) # Prints "5"
print f(6) # Prints "8"
g = make_adder(4)
print f(3) # Prints "7"
print f(6) # Prints "10"
```

LAMBDA NEVER HAS A RETURN

Generator

Uses the **yield** construct and the object method **next**.

Allows you to get a sequence of objects in a dedicated routine.

```
def countdown(n):
      while(n > 0):
            yield n
            n -= 1
# Creates the function call as object but does NOT run it yet
x = countdown(3)

print x.next() # First runs "countdown(3)" then prints "3"
print x.next () # Prints "2"
print x.next () # Prints "1"
```

Coroutine

Uses the **yield** construct and the object method **send** and **next**.

Allows you to pass a sequence of values one at a time to a function (e.g. log file printer)

```
def print_matches(text):
      print "Trying to find text: " + text
      while(True):
            line = (yield)
            if(text in line ):
                  print line

# Creates the function call as object but does NOT run it yet
x = print_matches("hello")
x.next() # Runs to first yield.
print  x.send("lalalala") # Prints nothing
print  x. send ("hello world") # Prints "hello world"
```

Classes

```
class ClassName(inherited_class1, inherited_closs2):
      # Class variables
      class_name = "Class Name"

      # Constructor
      def __init__(self):
            self.attribute1 = 1
            self.attribute2 = [3, 4]
            self.length_value = 1

      # Called without parenthesis for methosd
      @property
      def length(self)
            return self.length_value

      # Called by ClassName.static_method(arg)
      @staticmethod
      def print_class_name()
            print class_name
```

Calling Supercass Methods

Option #1
super(SuperClassName, **self**).methodName(variables)
Option #2
_ClassName__method_name(variables)

Invoking a Class Constructor: Use the class name followed by two parenthesis. Example for class "Stack":

Example: my_stack = **Stack()**

Class Special Methods:
__name__ Always preceded and proceeded by two underscores.

@property: Class methods that do not require parenthesis when called. Typically return an object or primitive.

Static Method: @staticmethod
Called using the **class name** not an object name.
Example:
ClassName.static_method()

Inheritance and Classes: Python class can inherit multiple classes.

Class and Inheritance Functions:

- **type**(variable_name): Returns a formatted string of object's class name.
- **isinstance**(variable_name, ClassName): Returns True if variable is of type ClassName, False otherwise.

Example: *isinstance*(my_stack, Stack) returns True.

- **issubclass**(SubclassName, ClassName): Returns true if SubclassName is a subclass of ClassName.

Example: *issubclass*(Stack, object) returns True.

Abstract Classes

Requires the import:
from abc **import** ABCMeta, **abstractmethod**, **abstractproperty**

```
# Required first line for abstract class
__meta_class__ = ABCMeta

@abstractmethod
def my_method(args):
    pass

@abstractproperty
def my_method(args):
    pass
```

Abstract classes do NOT inherit ABCMeta.

Exceptions

Format for an Exception

```
try:
      pass
except ErrorTypeName as error_object:
      # Catches only error of type ErrorName
      pass
except:
      # Catches all exceptions
      pass
finally:
      # Always run
      pass
```

Creating Your Own Exception

```
class MyException(exception):
      def __init__(self, errno, msg):
            self.args = (errno, msg)
            self.errno = errno
            self.msg = msg

class MyException2(exception):
      pass
```

Throwing an Exception
Use the **raise** keyword

```
raise MyException(404, "Access Forbidden")
```

Modules, Importing, and the sys Toolset

Importing From a Module with Normal Namespace

Syntax: **import filename**
Filename is the python filename without the file extension (.py). When importing in this fashion, it uses the file name as the namespace for the functions/classes in that file.

Example: Python file div.py has a function called divide that divides to integers.

```
import div
print div.divide(4,2)
```

Importing From a Module with a New Namespace

Syntax: **import** filename **as namespace**
Use a custom namespace name for

Example: Python file div.py has a function called divide that divides to integers. New namespace is named "foo"

```
import div as foo
print foo.divide(4,2)
```

sys – Common System Functions

```
import sys
```

Command Line Arguments:
sys.argv

Quitting Python:
sys.exit(0)

Printing to the Console (Substitute for print):
sys.stdout("Hello World")

Getting User Input from the Console:
input = **sys.stdin.readline()**

Function to Add Set of Integers Passed by Command Line

```
import sys

def sum_command_line_args()
    input_args = sys.argv
    sum = 0
    try:
        # Skip element one since module name
        for i in range(1, len(input_args)):
            sum += int(input_args[i])
    catch:
        print "Input argument not an integer"
        sys.exit(0)

# Print the sum to the console.
print "The sum of the input arguments is: ",
print sum_command_line_args()
```

Documentation String

Documentation String: First statement of a module, class, or function.

Extracting Documentation String for a Function, Class, or Module:

Use the method **__doc__**

Example: A function exists called fact. To print its documentation string, call:

print fact.__doc__

Accessing Documentation String Outside a Python Program

Example: Function *fact* exists in module MyModule.py

Interpretative Mode:
```
import(MyModule)
help(MyModule.fact)
```

Command Line:
```
pydoc MyModule.fact
```

Unit Testing

Included in **Documentation String**.

Module Name: doctest
Unit Test Function Name: testmod()

Format:
>>> function_name(*args*)
result

Example:
```
def multiply(a, b):
    """
    >>> multiply (0, 1)
    0
    >>> multiply (2, 1)
    2
    >>> multiply (3, -1)
    -3
    """
    return a * b
```

Setting Up doctest in Supporting Modules

```
# Check to see if this module is main
if( __name__ == 'main'):
    # Import doctest module then run testmod()
    import doctest
    doctest.testmod()
```

Benefits of Python

Good string and list processing functionality which minimizes awkward additional coding.	Scripted/interpreted coding available for testing
Higher order function support (e.g. functions can take other functions as arguments)	Syntax is comparable to other languages.
Good set of built-in libraries.	Wide range of free libraries and projects to build off.
People outside AI use it so others can appreciate your code.	

Midterm Special Notes

Python:

1. Do not forget colons in Python code including after function definitions, for, while, and if statements.
2. Do not forget to call imports in Python code for modules such as math, random, and sys.
3. Printing a formatted string of numbers can be written:

```
print "%3d %0.2f" % (10, .9799) # Prints 10 with a preceding space and 0.98
```

4. It is possible to have Tuples of size 0 by doing:

```
x = ()
```

5. It is possible to have Tuples of size 1 by doing:

```
x = "Hello World",
x = ("Hello World",)
```

6. For an abstract class, you need the line:

```
__metaclass__ = ABCMeta
```

General Agents:

7. **Components Needs to Pass the Turing Test:**
   a. **Natural Language Processing**
   b. **Knowledge Representation** (i.e. storage paradigm)
   c. **Automated Reasoning**
   d. **Machine Learning**
8. **Cognitive Science:** Brings together computer models from AI and experimental techniques from psychology to construct precise and testable **theories of the human mind.**
9. **Agent Function** – Maps percept sequence to agent action.
10. **Simple Reflex Agent** – Select actions based off the **current percept only.** Often defined by **condition-action rules** (i.e. **productions**)
11. **Goal Based Agents** – A **goal** is a binary condition (i.e. either met or not met). A goal based agent tries to reach a target goal. **Search and planning agents** may be goal based agents.
12. **Problem solving agents** deal with **atomic environments** (i.e. the environment is treated as a single whole and is **indivisible**).

Search:

13. In Recursive Best First Search code, remember to do the Goal_Test at the beginning of the function and to check if the successors list is empty after creating it.
14. Effective Branch Factor: b^* Equivalent branch factor if the search tree was modelled as a balanced tree (i.e. where the number of children for each node is equivalent for all nodes).

Constraint Satisfaction:

15. **Node Consistent Variable** – Any variable where every value in the variable's domain **satisfies all of its unary constraints** in a CSP network.
16. In AC-3, only excluding the current paired variable are expanded.
17. **Local Consistency:** Given a constraint graph, enforcing consistency (i.e. ensuring variables satisfy constraints) locally **in each part of the graph** leads to invalid values being eliminated throughout the graph.
18. **Path Consistency** – **A two variable set (X_i, X_j) are path consistent with respect to a third variable X_m** if for every assignment of values to X_i and X_j consistent with the constraint $\{X_i, X_j\}$, there is a valid assignment to X_m that satisfies the constraints $\{X_i, X_m\}$ and $\{X_m, X_j\}$.
19. **Interleaving Search and Inference** AC-3 can be used to infer reductions in the search domain both **before and during search**.
20. **Forward Checking** – One way to implement "Inference" in Backtracking algorithm. Whenever a variable is assigned, establish arc consistency for it on all unassigned variables. If arc consistency checking was done in preprocessing, forward checking adds no value.
21. **Minimum Remaining Value (MRV), Fail First, Most Constrained Variable Heuristic:** Select the variable to assign next that has the smallest inferred domain (i.e. least remaining legal values).

Logic and Logic Agents

22. Declarative Programming: Provide information to the agent on information it needs to know and it figures out how to achieve the solution. De Procedural approach: Teach the agent how to do certain actions and it uses that information to figure out a solution to what you intend for it to do.
23. **Background Knowledge** – Initial knowledge in the knowledge base.
24. **Inference** – Deriving new sentences from existing sentences.
25. **Logical Connectives:** Used to construct complex sentences out of atomic sentences.
26. **Theorem Proving:** Using sentences already in the model, apply **rules of inference** to construct a proof of the desired sentence without consulting models.
27. **Entailment Between Sentences: When one sentence logically follows from another sentence or set of sentences. It is similar to implies in philosophy.**
28. **Logical Inference:** **Process of drawing conclusions (i.e. new sentences) through entailment**. **Symbol of Inference:** $\vdash$ Given a knowledge base, *KB*, and a sentence α, if an inference algorithm, *i*, inferred α from *KB* then: $KB \vdash_i \alpha$
29. **Sound or Truth Preserving Inference Algorithm:** Can only **derive** entailed sentences. **Hence it cannot prove any sentence that is wrong.** **Example:** Model checking is a sound algorithm since it does not work on infinite spaces.
30. **Complete Inference Algorithm:** Can **derive** any entailed sentence. **A complete inference algorithm can prove anything that is right.**
31. **Literal:** In a complex sentence, a literal is either an atomic sentence (i.e. **positive literal**) or its negation (i.e. **negative literal**).
32. **Proof:** A chain of conclusions that leads to the establishing some statement following from the knowledge base.

General: $\sum_{i=m}^{n-1} a^i = \frac{a^m - a^n}{1-a}$

Inferences, Proofs, and Resolution

Three Key Notions in Propositional Logic

Logical Equivalence: $a \equiv b \Leftrightarrow (b \vdash a \wedge a \vdash b)$	**Validity** – A statement that is true in all models.	**Satisfiability** – A statement where at least one model can make the statement true.

Propositional Proof – A series of steps where each statement is either from the knowledge base, a valid propositional statement, or a statement follows previous statements via some rule of propositional inference.

Framing a Proof as a Search Problem

A propositional logic proof can be treated as search problem and existing search algorithms can be used to find a valid proof.	**Initial State:** The initial knowledge base	**Actions:** Set of all inference rules applied to all the sentences that match the first half of an inference rule	**Results:** Add the bottom half of all applicable inference rules (see actions) to the knowledge base.	**Goal:** A knowledge base that contains the statement that is trying to be proven.

Monotonicity – Property of some knowledge bases where the set of entailed sentences only increases as sentences are added to the knowledge base.	**Nonmonotonic logics** – Common in the study of human AI. Set of entailed sentences may decrease.	**Literal** – Propositional variables or their negation. Example: X or $\bar{X}$

Resolution

Resolution is a **sound** and **valid inference rule**.

Requires **two disjunctive clauses**. If the clauses contain complimentary variables, the two **clauses are combined with complementary literals excluded**.

Example of Resolution:

$$\frac{A \vee B \vee C, \quad \bar{C} \vee D \vee E}{A \vee B \vee D \vee E}$$

Resolvent: Clause produced by resolution. (i.e. bottom line of inference specifically: $A \vee B \vee D \vee E$)

Complementary Literals – One literal is the negation of the other literal.

Unit Resolution: Right hand clause contains a single literal whose complement is in the left clause.

Clause Set Notation: $\{L_1, L_2, \ldots, L_m\}$ is the same as a disjunction of those literals.

Conjunctive Normal Form (CNF): Conjunction (ANDs) of disjunctions (ORs).

Resolution works best on propositional knowledge bases in CNF.

Using CNF with Resolution

Goal: Prove $KB \Rightarrow \alpha$

Step #1: Use implication elimination

$$\overline{KB} \vee \alpha$$

Step#2: Negate the goal

$$KB \wedge \bar{\alpha}$$

Step #3: Convert to CNF

Step #4: Prove the statement is not satisfiable (i.e. the empty clause is found through resolution).

Truth Table Approach to Convert to CNF

- Enumerate all models.
- For any model that is false, take a disjunction of the literals negation.

Example:

A	B	Result
True	True	False
True	False	True
False	True	False
False	False	True

$$Result \Leftrightarrow (\bar{A} \vee \bar{B}) \wedge (A \vee \bar{B})$$

Inference Algorithm Approach to Convert to CNF

Key Inference Steps:

- **Double negation**
- **DeMorgan's Theorem**
- **Biconditional Elimination**
  $$(A \Leftrightarrow B) \Leftrightarrow \big((A \Rightarrow B) \wedge (B \Rightarrow A)\big)$$
- **Distributivity**
- **Implication Elimination**
  $$(A \Rightarrow B) \Leftrightarrow (\bar{A} \vee B)$$

Example:

$$(A \wedge B) \vee (\bar{A} \wedge \bar{B}) \vee (A \wedge \bar{B})$$
$$\neg\neg((A \wedge B) \vee (\bar{A} \wedge \bar{B}) \vee (A \wedge \bar{B}))$$
$$\neg((\bar{A} \vee \bar{B}) \wedge (A \vee B) \wedge (\bar{A} \vee B))$$
$$\neg((\bar{A} \vee \bar{B}) \wedge (A \vee B) \wedge (\bar{A} \vee B))$$
$$\neg\Big(\big((\bar{A} \wedge B\,) \vee (A \wedge \bar{B})\big) \wedge (\bar{A} \vee B)\Big)$$
$$\neg\Big(\big((\bar{A} \wedge B\,) \vee (A \wedge \bar{B})\big) \wedge (\bar{A} \vee B)\Big)$$
$$\neg(\bar{A} \wedge B)$$
$$(A \vee \bar{B})$$

Resolution Closure: Set of all statements that derive from the knowledge base through resolution.

Resolution Refutation Stops in Two Cases:

1. **Empty clause found**
2. **No new clauses are possible in the resolution closure.**

Refutation – Empty clause found when performing resolution.

Definite Clause – Disjunctive (OR) clause with **exactly one positive literal**.

Example: $(L \vee \bar{B} \vee \bar{C})$

Notation for Definite Clause:

$$PositiveLiteral\!: -NegativeLiterals$$

Example:

$$L\!: -B, C$$

ASCII Notation:

$$(B \wedge C) \Rightarrow L$$

Head: Positive literal in the clause (e.g. L)

Tail: Negative literals if any (e.g. B, C)

Rule: Entire clause.

Horn clause: Disjunctive clause with **at most one positive literal**.

Example Horn Clause: $\bar{B}$

Alternative Notation: $: -B$

Horn clause: Collection of Horn clauses. A type of **logic program**.

Importance of Horn Clauses and Program: Knowledge bases that are **Horn programs** can decide if a clause is entailed in **linear time and space**.

Goal: See if $KB \Rightarrow B$

Backward Chaining: If KB is a Horn program, look for a clause where B is the head. Check for a rule where the head is true. If one is found, then continue search.

Forward Chaining: If KB is a Horn program, start from the facts and **search forward until no possible change to KB or the goal is found.**

R_1	A (Fact)
R_2	C (Fact)
R_3	$\bar{A} \vee B$ (i.e. $A \Rightarrow B$)
R_4	$\bar{B} \vee \bar{C} \vee D$ (i.e. $(B \wedge C) \Rightarrow D$

Backward Chaining

Finds R_4 then R_3 then R_1 then R_2

Forward Chaining

Finds R_1 then R_2 then R_3 then R_4

Closed World Assumption (CWA) – Facts that are not known are assumed to be **false**. This favors **minimal models**.	**Open World Assumption (OWA)** – Facts that are not known are assumed to be **true**. This favors **maximal models**.

DPLL – Resolution Finding Algorithm

Three Optimizations Over the Basic Resolution Algorithm:

1. **Early Termination**: If all clauses are satisfied (have at least one positive literal) or any clause is false, terminate the algorithm.

2. **Pure Symbol Heuristic:** A **pure symbol** is any symbol that has the same sign in all clauses. Pure symbols are set to true if they exist.

3. **Unit Clause:** A **unit clause** contains on a single literal. The variable in the unit clause is set to true to satisfy the clause.

```
def DPLL_Satisfiable(s): # Returns True or False

    clauses = set of clauses from CNF representation of s
    symbols = list of symbols in s

    return DPLL(clauses, symbols, {})

def DPLL(clauses, symbols, model):
    # Check Early Termination
    if every clause is true in model:
        return True
    elif some clause is false in model:
        return False

    # Check Pure Symbol Heuristic
    P, value = FIND_PURE_SYMBOL(clauses, symbol, model)
    if P is not None:
        return DPLL(clauses, symbols – P, model U {P=value})

    # Check Unit Clause Heuristic
    P, value = FIND_UNIT_CLAUSE(clauses, model)
    if P is not None:
        return DPLL(clauses, symbols – P, model U {P=value})

    # Select first symbol and check both true and false
    P = FIRST(symbols)
    rest = REST(symbols)
    return DPLL(clauses, rest, model U {P = True})
        or DPLL(clauses, rest, model U {P = False})
```

Prolog

a. – Fact A in Prolog. b :- a – Horn Clause $(\neg a \vee b)$. Since a is true, then b is also true. c :- b – Horn Clause $(\neg b \vee c)$. Since b is true, then so is c d :- a, b – Horn Clause $(\neg a \vee \neg b \vee d)$. Since a and b are both true, so is d	This is the same as: a $a \Rightarrow b$ $b \Rightarrow c$ $(a \wedge b) \Rightarrow d$	Prolog supports non-Horn clauses like: $e{:}-not(a)$ and $f{:}-$false

Question #1 from Practice Final

$$\bigwedge_{i=1}^{6}\left(\overline{x_i} \vee \bigvee_{1\le j\le 6, i\ne j} x_j\right) \wedge \bigwedge_{i=1}^{6}\left(\bigwedge_{j=i+1}^{6}\left(\overline{x_i} \vee \overline{x_j} \vee \bigvee_{1\le k\le 6, k\ne i, k\ne j} x_k\right)\right) \wedge \bigwedge_{i=1}^{6}\left(\bigwedge_{j=i+1}^{6}\left(x_i \vee x_j \vee \bigvee_{1\le k\le 6, k\ne i, k\ne j} \overline{x_k}\right)\right) \wedge \bigwedge_{i=1}^{6}\left(x_i \vee \bigvee_{1\le j\le 6, i\ne j} \overline{x_j}\right)$$

First Order Logic

Logic based agents tell the knowledge base about their **percepts**.

First Order Logic – Logic system where variable domains is greater than solely "True" and "False"	**Variables:** Range over sets. **Usual notation:** x, y, z	**Constants:** Fixed values from a set **Usual notation:** a, b, c	**Function:** Take variables with function symbols and return a constant **Usual notation:** f, g, h,	**Predicate:** Takes inputs and outputs True/False **Usual notation:** P, Q, R

Term: A variable, a constant, or built up from these using function symbols and composition.	**Atomic Formula:** Predicate where each of the predicate slots is filled by a term. **Example:** $IsPrime(X * X + 3)$	**Formula:** An atomic formula or a composite of simpler formula.	**Universal Quantifier:** Symbol ∀ $\forall x F_1$ - For all x, F_1 is true.	**Existential Quantifier:** Symbol ∃ $\exists y F_2$ - For some y, F_2 is true.

First Order Logic Semantics

Universe (***M***) – A set M over which all variables range over.	**Constant** (c^M) – A value in the universe M	**Function** (f^M) – A Cartesian product defined as: $f^M: M * M * \ldots * M \rightarrow M$	**Predicate** (P^M): Returns True or false and is defined as: $P^M: M * M * \ldots * M \rightarrow \{T/F\}$	**Language:** Set of all constants in the universe and all function symbols.

Structure/Model (***M***): Combination of the universe, constants, functions, and predicates.	**Bound Variable:** A variable in a first order function that is within the scope of an existential or universal quantifier.	**Unbound Variable:** A variable in a first order function that has no quantifier.	**Example:** $(\exists x) F(x, y)$ **Unbound Variable:** y **Bound Variable:** x	**Variable/Object Assignment** (v): A map from unbound variables to elements in the universe (M)

Logic Equations with Quantifiers: $(\forall x)(\neg P) \Rightarrow \neg(\exists x)P$ $\neg((\forall x)P) \Rightarrow (\exists x)\neg P$	**Dealing with Predicates and Quantifiers:** $A(t) \Rightarrow (\exists x)A(x)$ (t is a term) $A(x) \Rightarrow (\forall y)(A(y))$	**Example:** Addition and Multiplication on Integers **Predicate:** $=^M$ **Functions:** $+^M$, $-^M$ **Model:** Includes set of natural numbers	**Not in Model:** $(\exists x)(1+1) * x = 1 + 1 + 1$ **In Model:** $(\exists x)(1+1) * x = 1 + 1 + 1 + 1$ x is 2

Interacting with a First Order Knowledge Base

TELL(KB, King(John)) – Tells the knowledge base the fact that John is a king. **TELL**(KB, Person(Richard)) – Tells the knowledge base that Richard is a person.	**Ask**(KB, King(John)) – Predicate that asks the knowledge base if John is a King. Would return true. **Ask**(KB, King(Zayd)) – Returns false since Zayd is not a king. This command is referred to as **query** or **goal**.	**AskVars**(KB, Person(x)) – Asks questions that returns a constant. Query response is known as a **binding list** or **substitution**. Example return is {x/Richard}	**Example First Order Knowledge Bases** 1. Any relational database 2. Basic set theory • No function symbols • = operator checks for equality • Constant is the empty set $\emptyset$

Theorem Proving in First Order Logic

Procedure 1. Convert all formulas in $KB \cup \{\neg\alpha\}$ into **prenex normal form**. Prenex normal form is: $\forall x \forall y \exists z\, F(x,y,z) \wedge G(x,y) \Rightarrow H(x,y,z)$ 2. **Skolemize the equation** to remove any existential quantifiers. 3. If all variables are bound and only universal quantifiers, **the quantifiers can be dropped and all variables are free**. 4. **Convert the open formula to CNF and use resolution to prove refutation**	**Skolemization Examples** $\exists x \exists y\, F(x) \Rightarrow G(y)$ skolemizes to $F(a) \Rightarrow G(b)$ $\forall x \forall y \exists z\, F(x,y,z) \Rightarrow G(x,y,z)$ skolemizes to $\forall x \forall y\, F(x,y,f(x,y)) \Rightarrow G(x,y,f(x,y))$	**Additional Notes** If there are only existential quantifiers, the variables are turned into constants and existential quantifiers dropped. To perform refutation, a substitution list may be required to ensure the terms in the predicate match. This can be checked using the **unification algorithm**.

Model checking is possible to prove entailment in first order knowledge bases. However, the time complexity is just as bad or worse than it is for propositional logic.

* = operator for checking two values are the same

First Order Logic Database Commands

PDDL – Planning Domain Definition Language
Successor of Strips language.
Planning – Application of first order logic. Develop a sequence of actions to achieve a goal while at each step in time satisfying all constraints.

Necessary Functions for Unify Function

- is_var(z) – Checks if z is a variable.
- is_term(z) – Checks if parameter z is a term.
- args(z) – Extracts a list of arguments in z
  - args((z*z)+35) – Returns (z*z, 35)
- op(z) – Gets the **outermost function symbol** in z
  - op((z*z)+35) – Returns "+"
- is_list(z) – Checks if parameter z is a list.
- head(z) – Returns first element in list z
- tail(z) – Returns all elements after the first element in z.

Necessary Functions for Unify_Var Function

- occur_ck(var, z) – Checks if z is function containing var
  - occur_ck(z, (z*z)+35) – Returns **True**
  - occur_ck(y, (z*z)+35) – Returns **False**
- append(new_sub, sub_list) – Appends the new substitution new_sub to the sub_list.

```
Unify(x, y, S):
    # x – a variable, constant, term, or list
    # y – a variable, constant, term, or list
    # S – substitution so far
    # returns a Substitution list or "None"

    # Check for previous failure
    if(S == None):
        return False
    # If with substitution the two parameters are the same
    # then return the substitution.
    if( x(S) == y(S)):
        return S
    # If x or y are variables, try to create a new substitution
    if(is_var(x)):
        return Unify_Var(x, y, S)
    elif(is_var(y)):
        return Unify_Var(y, x, S)
    elif( is_term(x) and is_term(y) ):
        return Unify(args(x), args(y), Unify(op(x), op(y), S) )
    elif( is_list(x) and is_list(y) ):
        return Unify( tail(x), tail(y), Unify( head(x), head(y), S) )
    else:
        return None
```

```
Unify_Var(var, y, S):
    # var – A variable
    # y – a variable, constant, term, or list
    # S – substitution so far
    # returns a Substitution list or "None"

    # Check if substitution exists for var (i.e. sub_val1)
    if( var |-> sub_val1 ):
        return Unify( sub_val1, y, S)
    # Check if substitution exists for y (i.e. sub_val2)
    elif( y |-> sub_val2 ):
        return Unify( var, sub_val2, S)
    # Check if y is a function f(var)
    elif( occur_ck(var, y) ):
        return None
    else:
        return append( var |-> y, S)
```

Unification Examples – These Can Be Simplified and To Just Unify Whatever Is After the "=" Signs.

Problem: Unify "x = f(z)" and "y = g(w)"

Step #1: Unify(x=f(z), y=g(w), {})
Step #2: Unify((x, f(z)), (y, g(w)), Unify(=, =, {}))
Step #3: Unify((x, f(z)), (y, g(w)), {})
Step #4: Unify((f(z)), (g(w)), Unify(x, y, {}))
Step #5: Unify((f(z)), (g(w)), Unify_Var(x, y, {}))
Step #6: Unify((f(z)), (g(w)), {x |-> y})
Step #7: Unify((), (), Unify(f(z), g(w), {x |-> y}))
Step #8: Unify((), (), Unify(z, w, Unify(f, g, {x |-> y})))
Step #9: Unify((), (), Unify(z, w, Unify_Var(f, g, {x |-> y})))
Step #10: Unify((), (), Unify(z, w, {x |-> y, f |-> g }))
Step #11: Unify((), (), Unify_Var(z, w, {x |-> y, f |-> g }))
Step #12: Unify((), (),{x |-> y, f |-> g, z |-> w }))
Step #13: Returns the substitution list {x |-> y, f |-> g, z |-> w }

Problem: Unify "x=[g(v), f(g(z))]" and "y=[g(f(w)), f(w)]"

Step #1: Unify(x=[g(v), f(g(z))], y=[g(f(w)), f(w)], {})
Step #2: Unify((x,[g(v), f(g(z))]), (y, [g(f(w)), f(w)]), Unify(=, =, {}))
Step #3: Unify((x,[g(v), f(g(z))]), (y, [g(f(w)), f(w)]), {})
Step #4: Unify(([g(v), f(g(z))]), ([g(f(w)), f(w)]), Unify(x, y, {}))
Step #5: Unify(([g(v), f(g(z))]), ([g(f(w)), f(w)]), Unify_Var(x, y, {}))
Step #6: Unify(([g(v), f(g(z))]), ([g(f(w)), f(w)]), {x |-> y})
Step #7: Unify((), (), Unify([g(v), f(g(z))], [g(f(w)), f(w)], {x |-> y}))
Step #8: Unify((), (), Unify([f(g(z))], [f(w)], Unify(g(v), g(f(w)), {x |-> y})))
Step #9: Unify((), (), Unify([f(g(z))], [f(w)], Unify(v, f(w), Unify(g, g, {x |-> y}))))
Step #10: Unify((), (), Unify([f(g(z))], [f(w)], Unify(v, f(w), {x |-> y})))
Step #11: Unify((), (), Unify([f(g(z))], [f(w)], Unify_Var(v, f(w), {x |-> y})))
Step #12: Unify((), (), Unify([f(g(z))], [f(w)], {x |-> y, v |-> f(w)})) (Exclude outer unify)
Step #13: Unify([], [], Unify(f(g(z)), f(w), {x |-> y, v |-> f(w)}))
Step #14: Unify([], [], Unify(g(z), w, Unify(f, f, {x |-> y, v |-> f(w)})))
Step #15: Unify([], [], Unify(g(z), w, {x |-> y, v |-> f(w)}))
Step #16: Unify([], [], Unify_Var(w, g(z), {x |-> y, v |-> f(w)}))
Step #17: Unify([], [],{x |-> y, v |-> f(w), w|->g(z)}))
Step #18: Functions return the substitution list: {x |-> y, v |-> f(w), w|->g(z)}

Planning

Problem Solving Agent – Goal based agent that is focused on solving problems on atomic domains.	**Planning Agents** – Goal based agents that work on factored domains.

PDDL – Planning Domain Definition Language

Heavily influenced by earlier planning languages including **STRIPS** and **ADL**.	**Fluent** – Facts that may change from situation to situation.	**Ground Fluent** – Fluent contain **no variable** (i.e. only constants). They are **functionless atoms**.	**State** – Conjunction of fluents that are ground. **States cannot contain negative atoms.**
Closed World Assumption – Fluents not in the knowledge base are false. (Used in PDDL)	**Unique Names Assumption** – Any objects that have different names are assumed to be different.	**Illegal Fluents in a State Description** 1. **Fluents containing variables.** Example: $At(x, y)$ 2. **Fluents containing negations.** Example: $\overline{Poor}$	Fluents are a **conjunction** so fluent **order does not matter**.

Actions in Planning

Actions need to clearly define what aspect of the state changes and what stays the same.	**Frame Problem:** In classical planning, most aspects of the state remain the same after an action. It can be **prohibitive to detail the countless aspect of a state that stayed the same after an action**	**Solution to the Frame Problem in PDDL: PDDL only enumerates the aspects of the state that change as a result of an action.** Any unmentioned aspects are assumed not to change.

PDDL Action Schema

Three Components in PDDL Action Schema 1. **Action Name and Input Variables** 2. **Precondition(s) if any** 3. **Effect(s)**	**Action Name and Input Variables** **Name** of the action performed and **any input variables**. **Example:** $Fly(p, from, to)$ **Action Name:** Fly **Variables:** p, $from$, to	**Preconditions** Aspects of the state that must be true before an action can be performed. **Cannot contain negated atoms.** **Example:** $At(p, from) \wedge Plane(p)$ $\wedge\, Airport(from)$ $\wedge\, Airport(to)$	**Effects** Action results. Changes in state. **Example:** $\neg At(p, From) \wedge At(p, To)$	**Complete Example** $\boldsymbol{Action}(Fly(p, from, to),$ $\boldsymbol{Precond}: At(p, from) \wedge Plane(p)$ $\wedge\, Airport(from)$ $\wedge\, Airport(to)$ $\boldsymbol{Effect}: \neg At(p, From) \wedge At(p, To))$

Applicable Action – An action a is applicable in state s if all of action a's preconditions are satisfied in state s.	In any given state, **multiple instances of a given action could be applicable**. Example: plane P_1 could fly from SFO to LAX or from SFO to JFK.	If an action has v variables and the variable have a maximum domain of size d, then it takes $\boldsymbol{O(v^d)}$ to find all applicable ground actions worst case.

Result of an Action – Conjunction of fluents.	**Delete List** – **Negative** literals in the result of an action. These negative literals correspond to fluents **deleted from the state**.	**Add List** – **Positive** literals in the result of an action. These positive literals correspond to fluents **added to the state**.	**Note: Actions do not refer to time**. Precondition refers to time t and results refer to time $t + 1$.

Planning Domain – Set of action schemas.	**Initial State** – **Conjunction of ground atoms.** Hence, every slot in the fluent must be filled.	**Goal** – Conjunction of Literals. **Goals can have variables**, which are treated as existentially quantified. **Goal Example:** $At(p, SFO) \wedge Plane(p)$ In this case, p could be any plane.	**Solution** – Sequence of actions from the initial state to a **state that ENTAILS the goal**.	**Inequality Condition** – Used to prevent illegal conditions in actions where two input variables have the same value.

Example Planning Algorithms

PlanSat – Given a planning problem, it determines whether a plan **exists** that solves the problem.	**BoundedPlanSat** – Given a planning problem, it determines whether a plan **exists** that solves the problem in **k steps or less**.	Both algorithms are **PSPACE** but **NP-Hard** (hard as any other problem in NP). For problems **without negative preconditions, PlanSet is polynomial time (P)**.

Planning as a Search Problem

Forward State Space Search – Start from the initial condition and search towards the solution.	**Backward (Regression) Relevant States Search** – Start from the goal and try to search backwards until a state IMPLIED by the start state is found. Referred to as **relevant-states search** since only states relevant to the goal are explored. At each step, there may be a **set of relevant states** (not just a single state).
Negatives of Forward State Space Search **Prone to search irrelevant states. Example:** Planning problem trying to go from $Buy(ISBN)$ and $OWN(ISBN)$. Would involve searching many irrelevant states. **Requires domain-independent heuristics since planning problems can have large state spaces.**	**Negatives of Forward State Space Search** **Partially uninstantiated actions and states** – Since a goal will not always detail a complete state, negative relevant states search often involves handling only partially instantiated actions and states. **It must also handle ground states.**

Heuristics in Planning – When trying to come up with a plan through search, heuristics may be helpful. **Example Heuristic Type** – Come up with plans to relaxed problems.	**Planning Problem and Search** **Nodes** – States in the state space. **Edges** – Actions in the planning domain (i.e. set of schemas)	**Solution** – Path (i.e. sequence of actions) to go from the initial state **to a state entailed by the goal state**.

Heuristics for Search

Ignore Preconditions Heuristic Drop all preconditions from actions. By itself, this is NOT an admissible heuristic as it may over estimate the solution. **Modified Approach** – Delete all effects except those literals that are in the goal. **Then count the number of actions needed to reach the goal.** When combined with the cost to get to the current node, **this heuristic allows you to use A* search** to find a plan. **Exact Count: NP-Hard since it does not reduce the number of states to search.** In **P time**, can a**pproximate the cost** within $log(n)$ factor where n **is the number of literals in the goal**	**Ignore Delete Lists** Remove all delete lists (i.e. set of negated literals) from all actions. Literals in the state are monotonically increase and if the goal is possible, it is eventually found. Still leaves a problem that is **NP Hard since it does not reduce the number of states to search.**	**State Abstraction** A many-to-one mapping from states in the ground representation of the problem to the abstract representation. **Example:** In the plane cargo problem, require that all packages have the same destination (e.g. a hub) and that packages can only start in one of five airports. **This usually entails ignoring some fluents.**

Decomposition – Key ideal in defining heuristics. It entails dividing a problem into parts, solving each part individually and then combining the parts. Similar to divide and conquer algorithm.	**Subgoal Independence Assumption** – Cost of solving a conjunction of subgoals is approximated by the sum of the costs to solve each subgoal independently. **This assumption can be optimistic or pessimistic**. Optimistic if when solving each subgoal, actions that would otherwise cancel each other do not. Pessimistic as there **may be redundant actions**.

Planning Graph

Planning Graph – Special data structure used to give better heuristic estimates for the cost of a plan. **Polynomial size approximation** of the tree one would get by exploring all actions. Useable for **propositional planning problems only**.	Level – Organizational structure for a planning graph. Each level is denoted as S_i • S_0 – **Initial State** Each level is linked by a set of possible actions. • A_0 – Set of all possible actions possible in level S_0	In each level, the set of achieveable literals are shown. **For a given S_i, both the positive (P) and negative ($\neg P$) could hold** given different sets of actions. S_i and A_i alternate in the tree. Action A_i: • Preconditions: S_i • Effects: S_{i+1}

Persistence Action – Type of no-op. Used to preserve/persist any literal which is not negated by an action. Every literal has a persistence action (small square in action) from S_i to S_{i+1} in the planning graph. **Once a literal appears in a level S_i it remains present for all future levels of the planning graph**.	**Mutex** – Mutual exclusion. Curved links to indicate things (e.g. actions, literals, etc.) that cannot occur at the same time.	**Leveled Off** – When two consecutive levels of a graph are identical. This is the **termination condition of the planning graph**.	Given graph with l literals and a actions: • $O(l)$ – Nodes maximum in each S_i • $O(l^2)$ – Mutex links in each S_i • $O(a+l)$ – Maximum number of nodes in each A_i • $O((a+l)^2)$ – Mutex links in each A_i • $O\big(2*(al+l)\big)$ – Effect and precondition links because each persistence action goes to one effect and one precondition link and every standard action a could go to l precondition and l effect links. Hence, for an n level planning graph, the maximum size is $\boldsymbol{O\big(n(a+l)^2\big)}$ **which is polynomial space. The construction time is equivalent.**

Using Planning Graphs for Heuristic Estimation

Unsolvable Planning Problem: Goal does not appear in the final step in the planning graph.	**Level Cost of $\mathbf{g_i}$** - First level in the planning graph where goal literal g_i first appears.	**Three Methods to Estimate Conjunction of Goals** 1. **Max-Level** 2. **Level-Sum** 3. **Set Level**	**Max Level** – Largest level cost amongst all the goal literals. **Admissible.**	**Level Sum** – Sum of the level costs of all goal literals. **Inadmissible**	**Set-Level** – Level in the graph **where all literals in the SET of goal literals first appear**. **Admissible.**

Graphplan

Graphplan – An algorithm that uses a planning graph to find a solution to a planning problem.	*(see code below)*
nogoods – Hash table containing level and goal combinations in the planning graph that failed to yield solutions. This prevents unnecessary repeat searching of the graph.	
INITIAL_PLANNING_GRAPH – Builds the planning graph for the initial state of the CSP (i.e. S_0).	
Conjuncts – Returns the goal statement as a conjunction of literals.	
Extract_Solution – Search through the planning graph to try to find the solution using either a constraint satisfaction approach or a search. **There are two common implementations of this function.**	
NumLevels – Number of levels in the planning graph	
Expand_Graph – Expand the graph to include action A_t, state S_{t+1}, and all mutex relations.	

```
def GraphPlan(problem): # Returns a solution or None
  graph = INITIAL_PLANNING_GRAPH(problem) # Build planning graph for initial state
  goals = CONJUNCTS(problem.GOAL)
  nogoods = {} # Empty hash table in Python

  for t = 0 to infty:
    # Only try to find a solution if the goal is achievable
    if all goals present and non-mutex in S_t of graph:
      solution = Extract_Solution(graph, goals, NumLevels(graph), nogoods)
      if(solution is not None):
        return solution
    # Termination found as graph has leveled off and no goods unchanged
    if graph and nogoods unchanged since t-1:
      return None
    graph = Expand_Graph(graph, problem)
```

Possible Implementations of the EXTRACT_SOLUTION Function

Extract_Solution as a Constraint Satisfaction Problem

Variables: Actions at each level of the graph. Hence a given action may appear as multiple different actions if it is present in multiple levels of the graph.

Domain: An action is either **IN** or **OUT** of the plan.

Constraints: Mutex relations (between literals and actions), goal literals, and preconditions.

Given this definition, any CSP solver can be used to find a plan if it exists.

Extract_Solution as a Backward Relevant State Search Problem

Each state in the planning graph contains a pointer to the previous level in the planning graph as well as a set of unsatisfied goals.

Search Criteria:

1. **Initial State** – S_n in the planning graph (since working backwards)
2. **Actions** – Set of actions A_{i-1} available in state S_i. User selects a **subset of conflict-free actions whose effects cover the goals in that state. Conflict free means actions which are not mutex and whose preconditions are not mutex.**
3. **Result –** A set of preconditions at state S_{i-1} based off the actions in A_{i-1} that must be fulfilled.
4. **Goal** – Reach S_0 with all goal literals satisfied
5. **Cost** – 1 for each action.

Level Cost – The level in the graph where a literal first appears. Example: Any literal in the initial state has a level cost of 0.	Graphplan is PSpace-Complete and making the planning graph can be done in polynomial time. Heuristics are needed to choose among actions in the planning graph.	**Graphplan Heuristic** 1. **Select the literal with the highest level cost** 2. **Prefer the action where the sum of the level costs of its preconditions is smallest.** This is a greedy based approach.

Types of Mutexes – Between Both Actions and Literals

Inconsistent Effects – One action negates the effect of another action.	**Interference** – One action's effect is the negation of the precondition of another action.	**Competing Needs** – One action's **precondition is mutually exclusive (not only negated) with the precondition** of another action.	**Inconsistent Support** – Two literals in a state **can only be achieve through mutually excluded actions**.

Practice Final Question #4

Variables • $foot_{Left}$ • $foot_{Right}$ • $sock_1$ • $sock_2$ • $shoe_{Left}$ • $shoe_{Right}$	**Predicates** • $Foot(foot)$ – Checks if $foot$ is a foot. • $Sock(sock)$ – Checks if $sock$ is a sock. • $Shoe(shoe)$ – Checks if $shoe$ is a shoe. • $Bare(foot)$ - Checks if $foot$ is bare. • $HasSockOn(foot)$ – Checks if $foot$ has on a $sock$ • $SockOff(sock)$ - Checks if $sock$ is off • $HasShoeOn(foot)$ - Checks if $foot$ has a shoe on. • $ShoeOff(shoe)$ – Checks if $shoe$ is off. • $SameFoot(foot, shoe)$ – Checks if $shoe$ and $foot$ are from same side (left/right)	$Action(\textbf{PutOnSock}(foot, sock),$ $\textbf{Precond}: Foot(foot) \wedge Bare(foot) \wedge Sock(sock) \wedge SockOff(sock)$ $\textbf{Effect}: \neg Bare(foot) \wedge HasSockOn(foot) \wedge \neg SockOff(sock))$ $Action(\textbf{PutOnShoe}(foot, Sock),$ $\textbf{Precond}: Foot(foot) \wedge HasSockOn(foot) \wedge \neg HasShoeOn(foot) \wedge Shoe(shoe) \wedge ShoeOff(shoe) \wedge SameFoot(foot, shoe)$ $\textbf{Effect}: HasShoeOn(foot) \wedge \neg ShoeOff(shoe))$	**Example Plan** $PutOnSock(foot_{Left}, sock_1)$ $PutOnSock(foot_{Right}, sock_2)$ $PutOnShoe(foot_{Left}, shoe_{Left})$ $PutOnShoe(foot_{Right}, shoe_{Right})$

Knowledge Representation

Complex domains require **more general and flexible knowledge representation** paradigms than "toy" domains like the Wumpus World.	**Common items that need to represented** • **Events** • **Time** • **Physical Objects** • **Beliefs**	**Ontological engineering** – A field that studies the methods and methodologies for representing knowledge specifically in ontologies.	**Ontology** – A formal naming and definition of the types, properties, and interrelationships of the entities that exist for a particular domain or discourse. **Frameworks which can be used to represent facts about the world so they can be used by knowledge based agents.**

Upper Ontology – **Hierarchical ontology in an Object-Oriented like style**. The most general representation is at the top of the tree/hiearchy. This is further subdivided into more specific classifications. Not able to handle exceptions to rules well.	Most successful ontologies **are specific to a certain domain**. Example: Create an ontology for circuits so that theorem provers could be developed to check the circuits.		

Knowledge Reasoning Systems

Objects in the world are often group into categories. **Two ways to create categories** 1. **First Order Predicates** 2. **Objects**	**First Order Predicates** Predicate checks for membership of an object within a category. **Example:** A category of objects could be Basketballs. The first-order predicate to check if something is a basketball: $Basketball(b)$	**Reification** Creating a category as an object itself. Hence, $Basketballs$ is an object that all basketballs are a component of. **It turns a proposition into an object.** Hence: $Basketball(b) \Rightarrow b \in Basketballs$	**Subset/Subcategory/Subclass** – A category that is a subset of a parent class. The subclass inherits a set of features from the parent class. **Example:** $Basketballs$ is a subclass of $Balls$

Inheritance – Entails that all statements that are true about the parent class are true about the subclass as well. **Example:** If all $Food$ is edible, then if $Fruit$ is a subclass of $Food$, then all $Fruit$ is edible too.	**Exhaustive Decomposition** – Every object in an original set is assigned to a subcategory. **Partition** – An exhaustive decomposition where all subcategories are disjoint (i.e. non-overlapping)	**Taxonomy/Taxonomy Hierarchy** – Organizational structure for representing subclass relationships.

Facts Taxonomies Can State

An object is a member of a category. Example: $BB_9 \in Basketballs$	**A category is a subclass of another category.** Example: $Basketballs \subset Balls$	**All members of a category have some property.** Example: $(x \in Basketballs) \Rightarrow Spherical(x)$	**Members of a category can be recognized by some set of properties.** Example: $Round(x) \land Orange(x) \land Diameter(x) = 9.5'' \Rightarrow Basketball(x)$	**Category as whole has some properties.** Example: $Dogs \in DomesticatedSpecies$

Physical Composition

Physical composition is useful to represent knowledge of physical objects.	**PartOf** – Relation that categorizes objects by saying they are part of another object. **Example:** $PartOf(Bucharest, Romania)$	**Composite Object** – A representation of an object by asserting the existence of its parts and their relationships. **Example:** Define a $Biped$ as having two legs that are **attached** to a body.	**BunchOf** – A relation that forms a composite object of **definite parts but no structure**. Objects in the **bunch** are **parts of the object not elements in it.**

Relationship Between PartOf and BunchOf If an element is part of a category, then it is a **PartOf a bunch containing that category**. $\forall x[x \in s \Rightarrow PartOf(x, BunchOf(s))$	$\forall y[[\forall x, x \in s \Rightarrow PartOf(x, y) \Rightarrow PartOf(BunchOf(s), y)]$ **If all members of a set/object are part of a bunch y, then a bunch of that set is also a part of the larger bunch y.**	**Logical Minimization** – Define an object as the smallest one possible while still satisfying certain conditions.

Measurements

Lengths and **measures** are turned into abstract measure objects.	**Unit Functions** – Used to represent lengths/measurements in terms of a unit (e.g. inches, hours, dollars, etc.). Note these do not return a normalized value rather are essentially a representation of units. **Example #1:** $Length(L_1) = Inches(d) = Centimeters(2.54 \times d)$ **Example #2:** $[b \in Basketballs] \Rightarrow [Diameter(b) = Inches(9.5)]$	Measures in a knowledge system are not numbers, but they can be used for ordering using symbols such as: **>**.

Events

Event Calculus – A logical language that deals with time rather than situations. Event calculus reifies (i.e. groups) fluents and events. PDDL uses a **situation calculus** that cannot say anything except before and after events.	$\mathcal{T}$ **Predicate** – A new predicate that tests whether some point in time or during some interval in time. **Example:** $T(At(Shankar, Berkeley), t)$ A test of whether $Shankar$ is at $Berkeley$ at some point t.	**Specific events are part of an events category.** **Example:** Describing the event E_1 of Shankar flying from SF to LAX could be: $(E_1 \in Flyings) \wedge Flyer(E_1, Shankar) \wedge Origin(E_1, SF) \wedge Destination(E_1, LAX)$	**Time Interval** – Has a start and end time (t_1, t_2). Also can be used in Event Calculus in the same way as individual points in time t.

Event Calculus Fluents

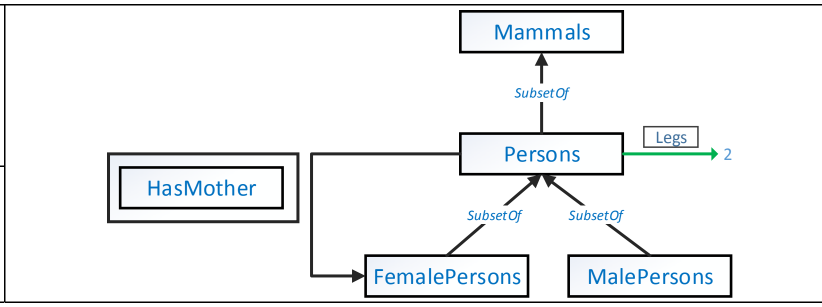
$T(f,t)$ - Predicate for whether **fluent f is true** at time or interval t	$Happens(f,t)$ - Predicate for whether fluent f **happened over** interval t	$Initiates(e,f,t)$ - Predicate for whether event e caused **fluent f to start to hold** at time t.	$Terminates(e,f,t)$ - Predicate for whether event e caused **fluent f to stop holding** at time t.	$Clipped(f,i)$ – Predicate for whether **fluent f ceased to be true sometime** during interval i.	$Restored(f,i)$ – Predicate for whether **fluent f became true sometime** during interval i.

Semantic Networks

- Used to integrate reasoning with knowledge systems based on categories.
- Do not support n-ary relations.
- Used to perform inheritance reasoning on graphs to determine what properties an object has.

Notation

- **Objects/Categories** – Ovals or boxes
- **Connections Between *Categories* Themselves** – Labeled Links
- **Connections Between *Objects* in the Same Category** – Outlined Box
- **Default Values** – Arrows to nothing



Description Logics

	Three Primary Inference Tasks in a Description Logic		
Description Logics – First order logics geared toward making it easier to describe definitions and properties of categories. **Example Description Logic: CLASSIC**	**Subsumption** – Checking if one category is a subset of another. Often is P-time and involves unification.	**Classification** – Checking if one object belongs to a category.	**Consistency** – Checking whether the membership criteria of a category is satisfiable.

Syntax in CLASSIC

$And(Concept_1, Concept_2, \ldots)$ **Example:** $Bachelor = And(Unmarried, Adult, Male)$	$All(RoleName, Concept)$ **Example:** All of a person's daughters are married and Unemployed: $All(Daughters, And(Married, Unemployed)$	$AtLeast(Integer, RoleName)$ **Example:** $AtLeast(2, Daughters)$	$AtMost(Integer, RoleName)$ **Example:** $AtMost(3, Sons)$	***OR* is not possible in classic nor negation so it is weaker than first order logic.**
$Fills(RoleName, IndividualName_1, \ldots)$ Role name fills one of the individual names. Example: $Fills(Department, Physics, Math)$	$OneOf(IndividualName_1, \ldots)$ Selects one of the individual name objects. **It is a limited type of disjunction.**			

Reasoning with Default Information

Jumping To Conclusions – Assuming default values for an object to be true without verification and if it is later shown to be untrue taking it back.	**Nonmonotonic Logic** – Set of beliefs in the knowledge base does not grow overtime. Rather new evidence can change existing beliefs.	**Circumscription** – More powerful and precise version of the closed world assumption. Every particular predicate is assumed to be "as false as possible" for every object except those objects for which it is known to be true.	**Circumscribed Predicate** – A specific predicate that a **circumscribed reasoner** is allowed to assume is false unless it is known to be true. **Example:** $Abnormal_1(x)$ which entails every object is normal unless known otherwise.	Circumscription deals with **preferred models** of the knowledge base rather than the requirement of truth of all models. **A model is preferred to another if it has fewer abnormal objects.**

Default Logic – A type of non-monotonic logic. It is a formalism of **default rules** which have the form: $\dfrac{Prequisite : Justification}{Conclusion}$	**Example of a Default Rule:** $\dfrac{Bird(x) : Flies(x)}{Flies(x)}$ The conclusion $Flies(x)$ is true unless it is known the justification is false. This rule can be rewritten: $Bird(x) \wedge \neg Abnormal(x) \Rightarrow Flies(x)$	**Extension (S)** – Maximum set of consequences from the default theory/rules and the facts. **For a given set of default rules and facts, there may be multiple possible extensions.**	**Nixon Diamond Problem** **Fact:** $Republican(Nixon) \wedge Quaker(Nixon)$ **Consider Two Default Rules** $\dfrac{Republican(x) : \neg Pacifist(x)}{\neg Pacifist(x)}, \dfrac{Quaker(x) : Pacifist(x)}{Pacifist(x)}$ **Two Possible Extensions** **Extension #1:** $\{Republican(Nixon) \wedge Quaker(Nixon) \wedge Pacifist(Nixon)\}$ **Extension #2:** $\{Republican(Nixon) \wedge Quaker(Nixon) \wedge \neg Pacifist(Nixon)\}$ **Both have the same number of abnormal objects so neither is preferred. Using these extensions, you can derive the abnormal object in the default rules.**

Decision Theory and Decision Theory Agents

Previous agents dealt with the world assuming everything was either: true, false, or unknown.	**Rational Decision** – Dependent on the relative importance of various goals and the likelihood (and degree/extent to which) these goals can be achieved.	**Example:** $\forall p\ Symptom(p, Toothache) \Rightarrow Disease(p, Cavity)$ Not always true since you can have a toothache for reasons other than a cavity.	**Possible Solution:** $\forall p\ Symptom(p, Toothache) \Rightarrow Disease(p, Cavity) \vee Disease(p, GumDisease) \vee \ldots$ This solution can be prohibitive since not all causes may be known or there are too many to enumerate individually.

Decision Theory – Takes the utility of all possible agents and adds it to some calculation based on the probability of achieving each of the possible goals.

$$DecisionTheory = ProbabilityTheory + UtilityTheory$$

Preference – The extent to which the agents prefers certain goal states/**outcomes** to others.

Example: An agent may prefer coffee twice as much as tea.

Utility Theory – Used to represent and reason about preferences. Utility theory says that every state has a degree of usefulness (i.e. utility) to an agent and that the agent prefers states with high utility.

Maximum Expected Utility (MEU) – Agent should choose the action which yields the highest expected payoff among the available choices.

```
def DT_Agent(percept):
    # Returns an action
    persistent belief_state   # Probability belies about the current state of the world
    persistent actions        # Set of agent actions.

    # Update set of probabilities based on percept and set of available actions
    update belief_state based on action and percept

    for action_i in actions:
        calculate outcome_probability_i  based off action description and belief state

    select action with highest expected utility given outcome_probability and utility information

    return action
```

Probability

Sample Space (Ω) – Set of all possible worlds. In other words, they are the set of all things that could be world. Elements in the sample space are **mutually exclusive**. **Example:** Sample space for rolling two six sided dice is: (1,1), (1,2), … (2,1), …(6,6)	**Probability Model/Probability Distribution** – Associate a number, $P(\omega)$, between 0 and 1 with each element (ω) in the sample space (Ω) with the condition that: $\sum_{\omega\in\Omega} P(\omega) = 1$	Probabilistic assertions may not be about individual worlds. Rather, it may deal with sets of them. Example: Probability of the sum of a two dice roll equaling 11 entails the case of (6,5) and (5,6). **Events** – Set of all possible worlds where a corresponding proposition holds (e.g. rolling 11).

Unconditional or Prior Probability – Degree of belief that a proposition holds in the absence of any other probability. **Example:** $P(11)$ or $P(doubles)$ for two dice roll.	**Evidence** – Additional information that may reveal information about the probability of other events.	**Conditional Probability** – Probability factoring in evidence from events. It is defined as: $P(A\|B) = \frac{P(A\wedge B)}{P(B)}$ for $P(B) > 0$ **Example:** $P(11\|Dice_1 = 5)$ is: $\frac{P(11 \wedge Dice_1 = 5)}{P(Dice_1 = 5)} = \frac{\frac{1}{36}}{\frac{1}{6}} = \frac{1}{6}$	**Notation for Probability Theory** • **Variables** – Initial capital letter • **Value from Domain/Sample Space** – Initial lower case letter

Random Variable (X) – A function that maps elements (ω) from the sample space (Ω) to the set of real numbers. $X : \Omega \to \mathbb{R}$ **Example Random Variable:** Bet \$3 on whether a coin flip is heads or tails. The random variable could be: $X(HEADS) = 3$ $X(Tails) = -3$	**Domain Variable** – Enumerate possible elements in the state space. There are **DIFFERENT from random variables** as they may not map to the set of real numbers. **Example Domain Variable:** $Weather = \{cloudy, rainy, sunny, snowy\}$	**Joint Probability Distribution** – Probability distribution of the Cartesian product of two or more random variables. **Example:** $P(Cavity, Weather)$ Joint probability distributions **allow us to discuss probability for sentences involving AND ($\wedge$)**: **Example:** $P(Cavity = true \wedge Weather = sunny)$

Random Variable Classifications

Boolean Random Variable (also known as an **Indicator Random Variable**) – Random variable where **each point in the sample space is mapped to one of two values**.	**Discrete Random Variable** – Random variable where the sample space is finite or if the image of the random variable is a subset of the integers.	**Continuous Random Variable** – Usually has a domain that consists an infinite number of states and where the function is continuous on the domain. **Example:** Sample space could be points in a room and random variable could be the temperature at a point in degrees Celsius. Calculating **probabilities of continuous random variables usually involves computing an integral**.

Important Probability Functions

$P(A\|B) = \frac{P(A \wedge B)}{P(B)}$	$P(A) = 1 - P(\neg A)$	$P(A \vee B) = P(A) + P(B) - P(A \wedge B)$	

	$P(A) + P(\neg A) = 1$		

Random variables often have interrelated values. Example: Probability of a toothache, cavity, and dentist catching your gums are related.

	Toothache		¬Toothache	
	Catch	¬Catch	Catch	¬Catch
Cavity	0.108	0.012	0.072	0.008
¬Cavity	0.016	0.064	0.144	0.576

Joint Probability Distribution for Random Variables $Cavity$, $Catch$, and $Toothache$.

This approach is not scalable with large numbers of random variables as it **grows a rate of 2^n for n random variables**.

Marginal Probability – Probability of a single random variable or single random variable's state without dependence on other random variables.

Marginalization/Summing Out – Given a joint probability function, $P(Y, Z)$ of two random variables, Y and Z, the marginal probability of random variable Y is found by:

$$\vec{P}(Y) = \sum_{z \in Z} \vec{P}(Y, z)$$

Example: $\vec{P}(Cavity) =$
$\{0.108 + 0.012 + 0.072 + .008,$
$0.016 + 0.064 + 0.144 + .0576\}$

Note the resulting probability is a VECTOR.

Conditioning – Dependent on the conditional probabilities to find the marginal probability of Y through:

$$\vec{P}(Y) = \sum_{z \in Z} \vec{P}(Y|z)$$

Note the resulting probability is a VECTOR.

Conditioning Example

Example: Find the conditional probabilities $P(Cavity|Toothache)$ and $P(\neg Cavity|Toothache)$.

$$P(Cavity|Toothache) = \frac{P(Cavity \wedge Toothache)}{P(Toothache)}$$

$$P(\neg Cavity|Toothache) = \frac{P(\neg Cavity \wedge Toothache)}{P(Toothache)}$$

Both $P(Cavity|Toothache)$ and $P(\neg Cavity|Toothache)$ contain $\frac{1}{P(Toothache)}$. Hence this can be simplified to:

$$\alpha\big(P(Cavity \wedge Toothache) + P(\neg Cavity \wedge Toothache)\big) = 1$$

$$P(Cavity \wedge Toothache) = 0.108 + 0.012 = 0.12$$

$$P(\neg Cavity \wedge Toothache) = 0.016 + 0.064 = 0.08$$

Hence, α is:

$$\alpha(0.12 + 0.08) = 1$$
$$\alpha = 5$$

$$P(Cavity \wedge Toothache) = 0.6$$
$$P(\neg Cavity \wedge Toothache) = 0.4$$

Normalization Constant (α) – Used to simplify calculations and to ensure the results each the expected value (e.g. 1 for probabilities)

Independence – Random variable states do not affect one another. **Hence, joint probability distributions can be factored into separate disjoint distributions.**

When A and B are independent:
$$P(A|B) = P(A)$$

Bayes' Rule

Given two non-independent random variables A and B, then:

$$P(A \wedge B) = P(A|B)P(B)$$
$$P(A \wedge B) = P(B|A)P(A)$$

Hence:

$$P(A|B)P(B) = P(B|A)P(A)$$

Importance of Bayes' Rule: If you need to know $P(A|B)$, it is hard to find but you know, $P(B|A)$, you can use Baye's rule in combination with marginal probabilities to solve for $P(A|B)$

Example: 70% of people with meningitis have a stiff neck. Odds of meningitis are 1/50000 (0.00002) and the odds of a stiff neck are 1/100 (0.01). The probability of $P(M|SN)$ is:

$$P(M|SN) = \frac{P(SN|M)P(M)}{P(SN)} = \frac{0.7 * 0.00002}{0.01} = 0.0014$$

Learning

Learning – Process by which an **agent improves its performance on future tasks after making observations about the world**.	**Applications of Learning** 1. **Programmer could not predict all possible situations an agent could encounter.** 2. **Programmer cannot predict changes over time.** 3. **Programmer might not have any idea to program a solution to the same problem themselves.**	**Component Improvements and Available Learning Techniques Depend On** 1. **Component to be improved.** 2. **Prior knowledge the agent has.** 3. **The representation used for the data and the component.** 4. **Available feedback to learn from.**

Inductive Learning – Learning from a set of input/output pairs and generating a general function that governs those pairs. Input is usually a **vector of attribute values.**	**Deductive/Analytic Learning** – Start from a set of general rules and **derive things logically entailed from these general reules.**	**Unsupervised Learning** – Agent learns patterns from the input although no explicit feedback is supplied. **Example: Clustering** – Input examples are **grouped into potentially useful clusters**.	**Reinforcement Learning** – **Agent learns through a series of reinforcements** (rewards or punishments). **Example #1:** Lack of a tip at the end of a journey gives the taxi agent it did something wrong. **Example #2:** Winning for a chess playing agent is a reinforcement it did something right.

Supervised Learning

Supervised Learning – Agent observes input-output pairs and **learns a function that maps from the input to the output**.	**Semi-supervised Learning** – Given a few labeled examples, the agent must make what it can from a large set of unlabelled examples.	**Training Set** – Set of original N input-output pairs, which are defined as: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ These are generated by some unknown function, f, defined as: $y = f(x)$	**Hypothesis** – A learned function h **that approximates the unknown function** f

Test Set – Disjoint from the training set. **Used to test the quality of the hypothesis** function h	**Classification** – Learning problem where the output y **is a finite collection of values**.	**Regression** – When the **output y is always a number (often an infinite range)**	**Consistent Hypothesis** – Any hypothesis function h that agrees (i.e. is consistent) with all input-output pairs. A given set of data **may have multiple consistent hypotheses**.	**Ockham's Razor** – Always prefer the **simplest hypothesis**. **Definition of "Simplest" may vary.** **Example of a Simpler Hypothesis** – First order polynomial versus degree 7 polynomial.

Decision Trees

A supervised learning algorithm

Takes a vector of input attributes and returns a **single output value**.

Input attributes can be either continuous or discrete.

Focus of this class is on Boolean decision trees. Hence the outputs are either:

- **Positive Examples** return **true**
- **Negative Examples** return **false**

Leaf nodes correspond to the **decision tree's result**. **Internal nodes** corresponding to **one of the input attributes**.

Not all paths (branches) in a decision tree need to be the same length.

Important Pseudocode Functions and Methods
Plurality_Value(examples) – Returns the most common boolean result from the set of examples
Importance(attribute, examples) – Returns a value quantifying the importance of attribute for the set of examples.
DecisionTree() – Python style constructor for an object of class DecisionTree
add_branch(attribute_value, subtree) – Method to append a subtree to the tree with the edge having the value "attribute_value".

Initial Call: decision_tree = **Decision_Tree_Algorithm**(all_examples, all_attributes, {})

```
# Builds a decision tree
def Decision_Tree_Algorithm(examples, attributes, parent_examples):
    # examples – Remaining unclassified examples
    # attributes – Remaining attributes not yet in the tree
    # parent_examples - Set of all examples in this node's parent.

    # No examples match this classification so return most common value for set of parents
    if( len(examples) == 0 ):
        return PLURALITY_VALUE(parent_examples)
    # All examples agree so return the agreed upon classification
    elif( all examples have same classification ):
        return classification
    # Since no attributes remaining, take most common value from remaining examples
    elif( len(attributes) == 0): return PLURALITY_VALUE(examples)
    else:
        # Find the most important attribute
        A = argmax_(a in attributes)Importance(a, examples)

        # Create a new tree
        tree = DecisionTree()

        # Iterate through all attribute values.
        for v_k in A:
            subset_examples = { exs in examples and E.A == v_k }
            subtree = Decision_Tree_Algorithm( subset_examples, attributes – A, parent_examples)
            # Add the subtree to the tree
            tree.add_branch( v_k, subtree )
        return tree
```

Decision Tree Importance Function

Importance function in the decision tree algorithm selects the next attribute in the tree.	**Good attribute selections** result in example sets that contain **either only positive or only negative examples**.	**Bad attribute selections** result in example sets that have the same proportion of positive and negative examples.	**Information Gain** – Quantifies the quality of an attribute selection.

Entropy

Entropy $H(v)$ – Fundamental quantity in information theory. It is a measure of the uncertainty of a domain variable. **The higher the entropy, the higher the uncertainty.** $$H(v) = -\sum_{v_k \in V} \left(P(v) log_2(P(v))\right)$$	**Entropy of a Boolean Random Variable $B(q)$:** $$B(q) = -1 * (q * \lg(q) + (1-q) * \lg(1-q))$$ **Entropy of an Attribute in a Decision Tree:** $$H(Goal) = B\left(\frac{p_k}{p_k + n_k}\right)$$ where p_k is the number of positive examples and n_k is the number of negative examples.	**Information gain** is defined as: $$Gain = H(S_i) - H(S_{i+1})$$ For an attribute, A, in a decision tree, this simplifies to: $$Gain(A) = B\left(\frac{p}{p+n}\right) - Remainder(A)$$ $Remainder(A)$ is a weight sum of the entropy of each random variable and its likelihood of occurring: $$Remainder(A) = \sum_{v_k \in A} \left(\frac{p_k + n_k}{p+n} * B\left(\frac{p}{p+n}\right)\right)$$

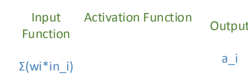
Neural Networks

Neurons – Type of brain cell. Electrochemical activity in the network of neurons is responsible for most mental activity.

Benefits of Neural Networks

1. **Perform distributed computation.**
2. **Tolerate noisy inputs**
3. **Learn**

Neural networks are composed of nodes or **units** called neurons.



Basic Neuron Structure

Each input link has a different weight as shown as the **thickness of the input arrow**.

Neuron Structure

A neuron is a link from unit i to unit j that propagates the **activation signal** a_i from i to j.

Note the activation signal is different than the activation function.

Weight ($w_{i,j}$) – Numeric value which determines the **strength and sign of the connection**.

Output of the Unit is derived from the **weighted sum function (in_j)** which is defined for **unit j** as:

$$in_j = \sum_{i=0}^{n} (w_{i,j} * a_i)$$

Activation Function (g_i) – From the weight function, it derives the neuron j's output (a_j). It is defined as:

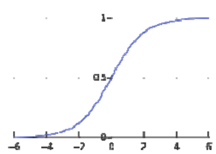
$$a_j = g(in_j) = g\left(\sum_{i=0}^{n} (w_{i,j} * a_i)\right)$$

Each neuron has a **single output that can be fed into several other neurons**.

Types of Activation Functions (g)

Threshold Activation Function – Output is binary (i.e. 0 or 1) depending on the weighted sum function's (in_j) value and the threshold value.

Logistic Function – A sigmoid curve in an "S" to mark the transition as more gradual.



Common Function for Logistic Function:

$$L = \frac{1}{1 + e^{-(x-x_0)}}$$

Perceptron – Uses a **threshold activation function** in the neural network's neurons.

Sigmoid Perceptron – Uses the **logistic function** in the neural network's neurons.

Feed-forward network – Neuron connections are only in a single direction. Hence, back connections are not permitted.

Recurrent Networks – Outputs of neurons are allowed to go back and serve eventually as its own input.

Can lead to oscillation in result but are more realistic.

Feed-forward networks are usually arranged into **layers** where **each layer only receives inputs from the previous layer**.

Hidden Layer – Any layer that is not connected to either an input or an output.

Perceptron/Single-Layer Neural Network – All inputs are connected to nodes whose outputs are the final outputs.

Neuron Weighted Sum Function: Given inputs x_1 and x_2 with activation signals a_1 and a_2 respectively, then the weight function for neuron 0 is:

$$w_0 + w_{1,0} * a_1 + w_{2,0} * a_2$$

Capabilities of a Feed-Forward Perceptron Network

1. Can calculate AND as well as OR
2. Cannot calculate XOR (parity) or binary summation.
3. **Generally can learn only linearly separable functions.**

There are some problems where a perceptron **will perform better than a decision tree (e.g. majority function)** while there are others **where the perceptron will perform worse (e.g. restaurant seating problem).**

Logistic Function – A type of function used as an activation function for a neuron. It has a general equal of the form:

$$LogisticFunction(t) = \frac{1}{1 + e^{-t}}$$

Logistic Function for Sigmoid Perceptron – The logistic function rewritten as a hypothesis function for a sigmoid perceptron is:

$$h_{\vec{w}}(\vec{a}) = LogisticFunction(\vec{w} \cdot \vec{a}) = \frac{1}{1 + e^{-\vec{w} \cdot \vec{a}}}$$

Value of Logistic Function Over Threshold Function:

The **logistic function is differentiable in the real number space** while a threshold function is not.

Loss Function in a Perceptron

Given a training set, E, with examples in the form $(\vec{x}, y)$ where y is a binary output (i.e. 0 or 1) and a hypothesis function $h_{\vec{w}}(\vec{x})$, then the **error or loss of $h_{\vec{w}}(\vec{x})$** is:

$$Loss(h_{\vec{w}}(\vec{x})) = \sum_{(x,y) \in E} (y - h_{\vec{w}}(\vec{x}))^2$$

Logistic Regression – Processing of fitting weights to a sigmoid perception.

Squaring in the loss function is used to prevent negative errors skewing the results.

Goal – Make the **loss/error function as close to 0 as possible**.

Finding the Loss Function for a Perceptron

$\frac{\partial}{\partial w_i} Loss(\vec{w}) = \frac{\partial}{\partial w_i}\left(y - h_{\vec{w}}(\vec{x})\right)^2$	Take the partial derivative with respect to one dimension (i.e. input to the perceptron) in the weight vector $\vec{w}$. Hence, if there is 10 inputs into the perceptron, then there is 11 elements in w (one for each input and one for the offset). This would require 11 partial derivatives.
$\frac{\partial}{\partial w_i} Loss(\vec{w}) = 2 * \left(y - h_{\vec{w}}(\vec{x})\right) \times \frac{\partial}{\partial w_i}\left(y - h_{\vec{w}}(\vec{x})\right)$	Derivate of $x^2 = 2x * \frac{dx}{dx}$. A similar implementation of the chain rule is followed here.
$\frac{\partial}{\partial w_i} Loss(\vec{w}) = 2 * \left(y - h_{\vec{w}}(\vec{x})\right) \times g'(\vec{w} \cdot \vec{x}) \times \frac{\partial}{\partial w_i}(\vec{w} \cdot \vec{x})$	y is a constant so its derivative is 0. Define $g'(\vec{x})$ as the derivative of $h_{\vec{w}}(\vec{x_i})$ which is the activation function g.
$\frac{\partial}{\partial w_i} Loss(\vec{w}) = 2 * \left(y - h_{\vec{w}}(\vec{x})\right) \times g'(\vec{w} \cdot \vec{x}) \times \vec{x_i}$	The partial derivative of $\vec{w}$ is 1 resulting in the final equation.
$\frac{\partial}{\partial w_i} Loss(\vec{w}) = 2 * \left(y - h_{\vec{w}}(\vec{x})\right) \times g(\vec{w} \cdot \vec{x})(1 - g(\vec{w} \cdot \vec{x})) \times \vec{x_i}$	Given the logistic function $g(t) = \frac{1}{1+e^{-t}}$, then $g'(t) = g(t)\left(1 - g(t)\right)$
$\frac{\partial}{\partial w_i} Loss(\vec{w}) = 2 * \left(y - h_{\vec{w}}(\vec{x})\right) \times h_{\vec{w}}(\vec{w} \cdot \vec{x})(1 - h_{\vec{w}}(\vec{w} \cdot \vec{x})) \times \vec{x_i}$	The activation function g is also known as $h_{\vec{w}}$.

Perceptron Update Rule

Perceptron Update Rule For each element in the training set $(\vec{x}, y)$, $newW_i$ is calculated for each dimension in $\vec{w}$. For the next training example to be considered, $newW_i$'s become the w_i's.	$newW_i = w_i + \alpha\left(y - h_{\vec{w}}(\vec{x})\right)h_{\vec{w}}(\vec{x})\left(1 - h_{\vec{w}}(\vec{x})\right)x_i$

Feed-Forward Learning

Feed-forward networks can compute more complicated networks than perceptron networks. **Example:** At most **a four level feed-forward network can create a spike** which is otherwise impossible with a perceptron network.	In a feed-forward network, the **activation function of the output unit is a composite of the activation function of many other units**.	**Example:** unit 5 has as inputs units 3 and 4. Unit #3 and #4 both have as inputs the initial inputs 1 and 2 (i.e. inputs to the network). $a_5 = g(w_{05} + w_{35}a_3 + w_{45}a_4)$ This can be rewritten as: $a_5 = g(w_{05} + w_{35}g(w_{03} + w_{13}x_1 + w_{23}x_2) + w_{45}g(w_{04} + w_{14}x_1 + w_{24}x_2))$	Hence, to solve for the individual weights requires **nonlinear regression**.

In back propagation, the values of a_k can be found be expanding back to the inputs as shown under "Feed-Forward Learning".

After minimizing the error at the output, the error is driven back in the network. This is through a process called **back propagation**.

k is the error of the k

in_k is the dot product of the weights and inputs into neuron k (i.e. $\vec{w} \cdot \vec{x}$).

Practice Final Questions

1. Mod3(x_1, ..., x_n) is the propositional formula which returns true if the number of variables `x_i` which are true in a truth assignment is exactly 0 mod 3. Write down a CNF formula for Mod3 in the case where `n=6`.

This uses the truth table\Karnaugh map approach to solve the problem. In the truth table, any assignment that makes the result false is added to the CNF as single clause that is the disjunction of the literals in the assignment but negated.

$$\bigwedge_{i=1}^{6}\left(\overline{x_i} \vee \bigvee_{1\le j\le 6, i\ne j} x_j\right) \wedge \bigwedge_{i=1}^{6}\left(\bigwedge_{j=i+1}^{6}\left(\overline{x_i} \vee \overline{x_j} \vee \bigvee_{1\le k\le 6, k\ne i, k\ne j} x_k\right)\right) \wedge \bigwedge_{i=1}^{6}\left(\bigwedge_{j=i+1}^{6}\left(x_i \vee x_j \vee \bigvee_{1\le k\le 6, k\ne i, k\ne j} \overline{x_k}\right)\right) \wedge \bigwedge_{i=1}^{6}\left(x_i \vee \bigvee_{1\le j\le 6, i\ne j} \overline{x_j}\right)$$

2. Give the DPLL algorithm and explain each of the three main "shortcuts" it checks for.

DPLL – Resolution Finding Algorithm

Three Optimizations Over the Basic Resolution Algorithm:

1. **Early Termination**: If all clauses are satisfied (have at least one positive literal) or any clause is false, terminate the algorithm.
2. **Pure Symbol Heuristic:** A **pure symbol** is any symbol that has the same sign in all clauses. Pure symbols are set to true if they exist.
3. **Unit Clause:** A **unit clause** contains on a single literal. The variable in the unit clause is set to true to satisfy the clause.

```
def DPLL_Satisfiable(s): # Returns True or False

    clauses = set of clauses from CNF representation of s
    symbols = list of symbols in s

    return DPLL(clauses, symbols, {})

def DPLL(clauses, symbols, model):
    # Check Early Termination
    if every clause is true in model:
        return True
    elif some clause is false in model:
        return False

    # Check Pure Symbol Heuristic
    P, value = FIND_PURE_SYMBOL(clauses, symbol, model)
    if P is not None:
        return DPLL(clauses, symbols – P, model U {P=value})

    # Check Unit Clause Heuristic
    P, value = FIND_UNIT_CLAUSE(clauses, model)
    if P is not None:
        return DPLL(clauses, symbols – P, model U {P=value})

    # Select first symbol and check both true and false
    P = FIRST(symbols)
    rest = REST(symbols)
    return DPLL(clauses, rest, model U {P = True})
        or DPLL(clauses, rest, model U {P = False})
```

3. (a) Let `x:= f(z)` and `y:= g(w)` explain how the unification algorithm from class would work on these inputs. (b) Now suppose `x:= [g(v), f(g(z))]` and `y:= [g(f(w)), f(w)]`. Explain how the unification algorithm from class would work on these inputs

Step #1: Unify("f(z)", "g(w)", {}) **Step #2:** Unify("z", "w" , Unify("f", "g", {})) **# Remove operator** **Step #3:** Unify("z", "w" , Unify_Var("f", "g", {})) **# Operators become variables** **Step #4:** Unify("z", "w" , { f |-> g }) **# Return of Unify_Var** **Step #5:** Unify_Var("z", "w" , { f |-> g }) **# Unify the variables** **Step #6:** { f |-> g, z |-> w } **# Final Substitution**	**Step #1:** Unify("[g(v), f(g(z))]", "[g(f(w)), f(w)]", {}) **# Remove the head of the lists.** **Step #2:** Unify("[f(g(z))]", "[f(w)]", Unify("g(v)", "g(f(w))", {})) **# Remove outermost function symbols g.** **Step #3:** Unify("[f(g(z))]", "[f(w)]", Unify("v", "f(w)", Unify("g", "g", {})) **# No unification required since function operators are identical** **Step #4:** Unify("[f(g(z))]", "[f(w)]", Unify("v", "f(w)", {}) **# Unify on variable v** **Step #5:** Unify("[f(g(z))]", "[f(w)]", Unify_Var("v", "f(w)", {}) **# Append to substitution list for variable v** **Step #6:** Unify("[f(g(z))]", "[f(w)]", { v |-> f(w) }) **# Extract the first item in each list.** **Step #7:** Unify("[]", "[]", Unify("f(g(z))", "f(w)", { v |-> f(w) })) **# Extract function symbol f on the two functions** **Step #8:** Unify("[]", "[]", Unify("g(z)", "w", Unify("f", "f", { v |-> f(w) }))) **# Unify on identical function symbols f** **Step #9:** Unify("[]", "[]", Unify("g(z)", "w", { v |-> f(w) })) **# Perform Unify var on variable w** **Step #10:** Unify("[]", "[]", Unify_Var("w", "g(z)", { v |-> f(w) })) **# Append substitution list for variable w** **Step #11:** Unify("[]", "[]", "w", "g(z)", { v |-> f(w), w|-> g(z) }) **# Identical unification lists so no step here** **Step #12:** { v |-> f(w), w|-> g(z) } **# Final Substitution**

4. Consider the problem where you have two socks and two shoes all of which are on the ground. You also have two feet. Your goal is to put on your shoes. Your feet can wear socks, but not shoes directly. Your available actions are to put on socks and put on shoes. Formulate this problem reasonably in PDDL. Then give an example plan solving it.

Predicates		Example Plan
1. $Shoe(shoe)$ - Returns whether "$shoe$" is a shoe. 2. $Sock(sock)$ – Returns whether "$sock$" is a sock. 3. $Foot(foot)$ - Returns whether "$foot$" is a foot. 4. $Bare(foot)$ - Returns whether "$foot$" is bare (i.e. has no socks or shoes) 5. $HasSock(foot)$ - Returns whether "$foot$" has a sock on already. 6. $HasShoe(foot)$ - Returns whether "$foot$" has a shoe on already. 7. $SockIsOff(sock)$ - Returns whether "$sock$" is off. 8. $ShoeIsOff(shoe)$ - Returns whether "$shoe$" is off. 9. $SameFoot(foot, shoe)$ - Returns whether "$foot$" and "$shoe$" go on the same side (e.g. left or right) **Constants** Foot: $foot_{Left}, foot_{Right}$ Sock: $sock_1, sock_2$ Shoe: $shoe_{Left}, shoe_{Right}$	$Action(PutOnSock(foot, sock),$ $Precond: Foot(foot) \wedge Sock(sock) \wedge Bare(foot) \wedge SockIsOff(sock)$ $Effect: \neg Bare(foot) \wedge HasSock(foot) \wedge \neg SockIsOff(sock))$ $Action(PutOnShoe(foot, shoe),$ $Precond: Foot(foot) \wedge Shoe(shoe) \wedge HasSock(foot)$ $\wedge \neg HasShoe(foot) \wedge ShoeIsOff(shoe),$ $Effect: HasShoe(foot) \wedge \neg ShoeIsOff(shoe))$	$PutOnSock(foot_{Left}, sock_1)$ $PutOnSock(foot_{Right}, sock_2)$ $PutOnShoe(foot_{Left}, shoe_{Left})$ $PutOnShoe(foot_{Right}, shoe_{Right})$

5. Show how the Graphplan algorithm would work on the example of the previous problem.

6. Define the following terms related to knowledge engineering: (a) ontology, (b) reification, (c) taxonomy.

Ontology – A formal naming and definition of the types, properties, and interrelationships of the entities that exist for a particular domain or discourse. It is a framework **which can be used to represent facts about the world so they can be used by knowledge based agents.**

Reification is the process of turning a predicate into an object. For example, all basketballs are reified into an object $Basketball$ such that $\forall b[Basketball(b) \Rightarrow b \in Basketballs$.

Taxonomy/Taxonomy Hierarchy – Organizational structure for representing subclass relationships.

7. Explain and give an example of the following concepts from probability theory: (a) random variable, (b) marginalization, (c) Bayes' rule.

A **random variable** (X) maps elements in the state space (Ω, i.e. the set of possible, disjoint worlds), to the set of real numbers. Hence: $X : \Omega \rightarrow \mathbb{R}$

Example: We bet $3 on the result of a coin flip. A random variable X could be:

$$X(HEADS) = 3$$
$$X(TAILS) = -3$$

Marginalization is the extraction of probability of a single random variable from a joint probability distribution function.

$$\vec{P}(Y) = \sum_{z \in Z} \vec{P}(Y, z)$$

Note these are vectors.

Example:

	Toothache	¬Toothache
Cavity	0.1	0.2
¬Cavity	0.3	0.4

$$P(Cavity) = \{0.1 + 0.2, 0.3 + 0.4\} = \{.3, .7\}$$

Bayes' Rule comes from conditional probability which is defined as P(A) given B or:

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

Using $P(A \wedge B)$, it can be shown:

$$P(A|B)P(B) = P(B|A)P(A)$$

Example: Probability of a stiff neck if you have meningitis is 0.7. Probability of meningitis is $\frac{1}{50000}$ and the probability of a stiff neck is 0.01. Hence the probability you have meningitis given a stiff neck is:

$$P(M|SN) = \frac{P(SN|M)P(M)}{P(SN)} = \frac{0.7 * 0.00002}{0.01} = 0.0014$$

8. Consider the following training set of 4-tuples.

(T,T,T, F)
(T,T,F, F)
(T,F,T, F)
(T,F,F, F)
(F,T,T, T)

Here `T` is short for true, `F` is short for false. The first three columns correspond to the variables `x_1`, `x_2`, `x_3`, the last column is the output of some function `f`. Calculate `Gain(x_i)` for `i=1,2,3`. Which variable should we use as the top of a decision tree for `f`?

To calculate the information gain for each parameter, you only need to calculate the $Remainder(x_i)$ of each attribute x_i. The attribute with the lowest $Remainder$ is the one to selected. Hence:

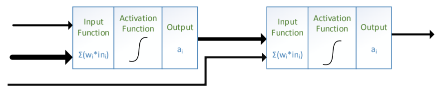
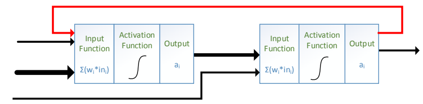
$$Remainder(x_1) = \frac{1}{5} * B\left(\frac{1}{1}\right) + \frac{4}{5} * B\left(\frac{0}{4}\right)$$
$$Remainder(x_1) = 0$$

$$Remainder(x_2) = \frac{2}{5} * B(0) + \frac{3}{5} B\left(\frac{1}{3}\right)$$
$$Remainder(x_2) = 0 + \frac{3}{5} B\left(\frac{1}{3}\right)$$
$$Remainder(x_2) = -\frac{3}{5} * \left(\frac{1}{3} \lg\left(\frac{1}{3}\right) + \frac{2}{3} \lg\left(\frac{2}{3}\right)\right)$$
$$Remainder(x_2) > 0$$

$$Remainder(x_3) = \frac{2}{5} * B(0) + \frac{3}{5} * B\left(\frac{1}{3}\right)$$
$$Remainder(x_3) = 0 + \frac{3}{5} B\left(\frac{1}{3}\right)$$
$$Remainder(x_3) = -\frac{3}{5} * \left(\frac{1}{3} \lg\left(\frac{1}{3}\right) + \frac{2}{3} \lg\left(\frac{2}{3}\right)\right)$$
$$Remainder(x_3) > 0$$

Since x_1 has the lowest remainder, it is the attribute that should be expanded at the top of the tree. This also makes intuitive sense as x_1 results in sets containing only positive or only negative examples.

9. Give the formal definition of a perceptron. Explain and give an example of a feed forward network is and what a recurrent network is.

A **perceptron** is a neural network where the activation function (g) is exclusively a threshold function (e.g. 0 if below the threshold and 1 if above or equal to the threshold). It cannot have a logistic function as its activation function which has a more gradual turn-on profile. Such networks are called **sigmoid perceptrons**. A perceptron network is a single layer networks meaning the inputs are connected to **units** that are exclusively connected to final outputs.	A **feed-forward network** is a neural networks where the outputs neuron's only move in a single direction (i.e. forward). No back lines are allowed in the network so a neuron's output can never form part of its own input signals. **Example of a Feed-Forward Network With a Single Neuron** 	A **recurrent network** is a neural networks where the outputs of neurons are looped back to eventually form part of the neuron's inputs (either directly or through a predecessor node). **Example of a Recurrent Network With a Single Neuron** 

10. Give and explain the update rule for learning neuron weight from class.