# 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.<br><br>**Components Needs to Pass the Turing Test:**<br>1. **Natural Language Processing**<br>2. **Knowledge Representation** (i.e. storage paradigm)<br>3. **Automated Reasoning**<br>4. **Machine Learning** | **Total Turing Test** – A variant of the Turing Test where the robot passes entirely as a human.<br><br>**Additional Requirements Over Standard Turing Test:**<br>1. **Computer Vision**<br>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.<br><br>**The maximizing action depends on:**<br>1. **Performance Measure**<br>2. **Any prior/built-in knowledge of the agent**<br>3. **Percept sequence to date.**<br>4. **Set of possible actions.** | **Percept** – An agent's perceptual inputs through **sensors** at any given instant.<br><br>**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**<br>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**<br>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**<br>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**<br>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**<br>Does the environment **change while the agent is making a decision**? Chess is static while taxi driving is dynamic. | **Discrete vs. Continuous**<br>Time, percepts, and actions divided **into a fixed, finite set** (e.g. chess)? A continuous environment is taxi-driving. | **Known versus Unknown**<br>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<br>**Edge/Branches**: Actions<br>**Node/Vertices**: States in the state space<br>**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.<br>**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.<br>1. Incomplete if $d > l$<br>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).<br><br>2. Can be considered special case of depth-limited search with $l = \infty$<br><br>**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.<br><br>**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.<br><br>**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.<br>$C^*$ - Minimum (optimal) cost to the goal.<br>$\epsilon$ - 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**.<br>$f(n) = h(n)$<br>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:

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

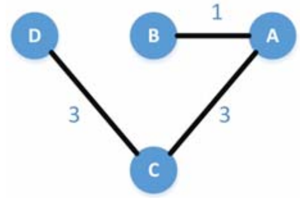
```
def UCS(problem):
    initial_state = problem.INITIAL_STATE()
    priority_queue = {}
    explored_set = {}
    priority_queue.enqueue(initial_state)

    # 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. |

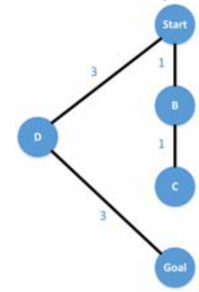
## Example of A* Performing Better than Greedy Best First Search




**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 $\infty$

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 $\mu$. 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:** $f(n) = c(n) + h(n)$<br><br>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'*.<br><br>$h(n) \leq c(n, a, n') + h(n')$<br><br>**Note:** Any heuristic that is consistent is also admissible.<br><br>**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**<br>If $h(n)$ was a consistent heuristic, then the values of $f(n)$ are nondecreasing.<br><br>**Given a node *n'* is a successor of *n* through action a, then:**<br><br>$g(n') = g(n) + c(n, a, n')$<br><br>If *h(n)* is consistent, then:<br><br>$h(n') + c(n, a, n') \geq h(n)$<br><br>Then:<br>$g(n') + h(n') + c(n, a, n') \geq g(n) + h(n) + c(n, a, n')$<br><br>$f(n') + c(n, a, n') \geq f(n) + c(n, a, n')$<br><br>$f(n') \geq f(n)$ | **Lemma #2:** Whenever A* selects a node for expansion, the optimal path to that node has been found.<br><br>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**<br><br>**By Lemma #2:** If a goal node is explored, it is the optimal path to that goal node.<br><br>**By Invariant of A*:** A* algorithm explores nodes in non-decreasing order of $f(n)$.<br><br>**By Lemma #1:** $f(n)$ is nondecreasing.<br><br>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.<br><br>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.<br><br>$N + 1 = 1 + b + b^2 + \cdots + b^d$<br>$\boldsymbol{N + 1 = \frac{b^{*d+1} - 1}{b^* - 1}}$<br>Derives from:<br>$\sum_{i=m}^{n-1} a^i = \frac{a^m - a^n}{1 - a}$<br><br>**Best branch possible factor is 1.** | **Relaxed Problem:** A version of the actual problem with fewer restrictions.<br><br>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.<br><br>**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:<br><br>$h_{composite} = \max\{h_1, h_2, \ldots, h_n\}$ | |

| | | | |
|---|---|---|---|
| **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.<br><br>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.<br><br>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.<br><br>**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.

```
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. |

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

**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.**

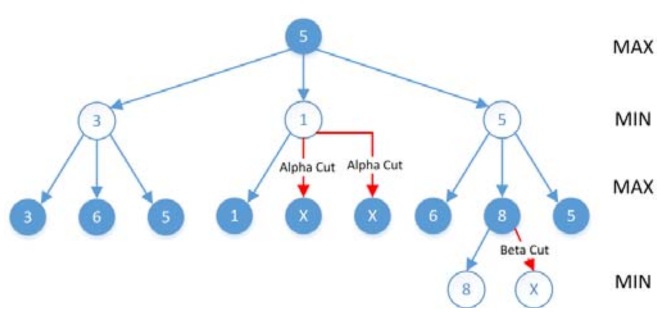
| | |
|---|---|
| **Time Complexity with Alpha-Beta Pruning:** $O\left(b^{\frac{d}{2}}\right)$ | **Time Complexity without Alpha-Beta Pruning:** $O(b^d)$ |

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

| | | | |
|---|---|---|---|
| **Assignment** – Allocation of values to variables.<br><br>**Solution:** A complete and consistent assignment. | **Consistent Assignment** – An assignment of values that does not violate any constraints.<br><br>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**<br>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).<br><br>**Simplest CSP Type:** Finite, discrete domain | **Constraint Language**<br><br>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).<br><br>Constraint functions can also be **nonlinear**. |

## Constraint Types

| | | | | |
|---|---|---|---|---|
| $X = \{X_1, X_2\}$ and $D = \{A, B\}$<br><br>**Example Constraint:**<br>$C = \langle (X_1, X_2), rel \rangle$<br>with<br>$rel = \{(A, B), \quad (B, A)\}$ | **Precedence Constraint:** A constraint that forces one variable to occur before (i.e. be less than) another variable.<br><br>**Example:**<br>$T_1 + d \leq T_2$ | **Disjunctive Constraint:** A constraint that two variables do not overlap (i.e. are not equal):<br><br>**Example:**<br>$T_1 + d \leq T_2$ 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.<br><br>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.<br><br>**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).<br><br>**Example:**<br>***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.<br>-1000 points for falling into a pit or eating a wumpus.<br>-1 for each action taken.<br>-10 for using an arrow. | Move forward one room.<br>Turn left 90 degrees.<br>Turn right 90 degrees.<br>Shoot the arrow<br>Climb out (if in starting space) | **Stench:** A wumpus is in an adjacent room.<br>**Breeze:** A pit is in an adjacent room.<br>**Glitter:** Gold is in the player's room<br>**Scream:** Wumpus is killed.<br>**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**.<br><br>**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.<br><br>**Example:** If sentence $\alpha$ is true in model $m$, then model $m$ **satisfies** sentence $\alpha$. |

## 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 $\alpha$ and $\beta$, then sentence $\alpha$ entails the sentence $\beta$ 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 $\alpha$. Hence:

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

**Model checking entails enumerating all possible models to determine whenever $KB$ is true that $\alpha$ 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 $\alpha$, if an inference algorithm, $i$, inferred $\alpha$ 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**"

**$A \Rightarrow B$ is True unless $A$ is true and $B$ is false. $A \Leftrightarrow B$ 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

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

### Operator Precedence

$$\neg,\ \vee,\ \wedge,\ \Rightarrow,\ \Leftrightarrow$$

# 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 $\alpha$ 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 $\alpha$ and $\beta$ that are true in the same set of models.<br>**Notation:** $\alpha \equiv \beta$ | **Validity**: A sentence that is **valid** (**true**) in **all models**.<br>**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 ∧** | $((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma))$ | **Associativity of ∨** | $((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma))$ |
| **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** | $(\alpha \Leftrightarrow \beta) \equiv ((\alpha \wedge \beta) \vee (\neg \alpha \wedge \neg \beta))$ |
| **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 ∨** | $(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$ | **Distributivity of ∧ and ∨** | $(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$ |
| **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.<br><br>**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.<br><br>**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**.<br><br>**Definition:**<br>$S_1$: Propositional symbol for the first switch and is true if the switch is on and false otherwise.<br>$S_2$: Propositional symbol for the second switch and is true if the switch is on and false otherwise.<br>$S_3$: Propositional symbol for the third switch and is true if the switch is on and false otherwise.<br>$S_4$: 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.<br>$\neg(S_1 \wedge S_4)$<br>$\neg S_1 \vee \neg S_4$<br><br>**Part B:** At least one switch must be on.<br>$S_1 \vee S_2 \vee S_3 \vee S_4$ |

# 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 $\alpha$, if an inference algorithm, *i*, inferred $\alpha$ 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:**

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

**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<br><br>**Three Optimizations Over the Basic Resolution Algorithm:**<br><br>1. **Early Termination**: If all clauses are satisfied (have at least one positive literal) or any clause is false, terminate the algorithm.<br><br>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.<br><br>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. | <pre>def DPLL_Satisfiable(s):<br>    # Returns True or False<br><br>    clauses = set of clauses from CNF representation of s<br>    symbols = list of symbols in s<br><br>    return DPLL(clauses, symbols, {})<br><br>def DPLL(clauses, symbols, model):<br>    # Check Early Termination<br>    if every clause is true in model:<br>        return True<br>    elif some clause is false in model:<br>        return False<br><br>    # Check Pure Symbol Heuristic<br>    P, value = FIND_PURE_SYMBOL(clauses, symbol, model)<br>    if P is not None:<br>        return DPLL(clauses, symbols – P, model U {P=value})<br><br>    # Check Unit Clause Heuristic<br>    P, value = FIND_UNIT_CLAUSE(clauses, model)<br>    if P is not None:<br>        return DPLL(clauses, symbols – P, model U {P=value})<br><br>    # Select first symbol and check both true and false<br>    P = FIRST(symbols)<br>    rest = REST(symbols)<br>    return DPLL(clauses, rest, model U {P = False})<br>        or DPLL(clauses, rest, model U {P = False})</pre> |

## Prolog

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

# 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.<br><br>**Usual notation:** x, y, z | **Constants:** Fixed values from a set<br><br>**Usual notation:** a, b, c | **Function:** Take variables with function symbols and return a constant<br><br>**Usual notation:** f, g, h, | **Predicate:** Takes inputs and outputs True/False<br><br>**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.<br>**Example:** $IsPrime(X*X+3)$ | **Formula:** An atomic formula or a composite of simpler formula. | **Universal Quantifier:** Symbol ∀<br><br>$\forall x F_1$ - For all $x$, $F_1$ is true. | **Existential Quantifier:** Symbol ∃<br><br>$\exists y F_2$ - For some $y$, $F_2$ is true. |

## First Order Logic Semantics

| | | | | |
|---|---|---|---|---|
| **Universe** ($\boldsymbol{M}$) – 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:<br>$f^M: M*M*\ldots*M \rightarrow M$ | **Predicate** ($P^M$): Returns True or false and is defined as:<br>$P^M: M*M*\ldots*M \rightarrow \{T/F\}$ | **Language:** Set of all constants in the universe and all function symbols. |

| | | | | |
|---|---|---|---|---|
| **Structure/Model** ($\boldsymbol{M}$): 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:**<br>$(\exists x)F(x,y)$<br>**Unbound Variable:** $y$<br>**Bound Variable:** $x$ | **Variable/Object Assignment** ($v$): A map from unbound variables to elements in the universe ($M$) |

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

## Interacting with a First Order Knowledge Base

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

## Theorem Proving in First Order Logic

| | | |
|---|---|---|
| **Procedure**<br>1. Convert all formulas in $KB \cup \{\neg\alpha\}$ into **prenex normal form**. Prenex normal form is:<br>$\forall x \forall y \exists z\, F(x,y,z) \wedge G(x,y) \Rightarrow H(x,y,z)$<br><br>2. **Skolemize the equation** to remove any existential quantifiers.<br><br>3. If all variables are bound and only universal quantifiers, **the quantifiers can be dropped and all variables are free**.<br><br>4. **Convert the open formula to CNF and use resolution to prove refutation** | **Skolemization Examples**<br>$\exists x \exists y\, F(x) \Rightarrow G(y)$<br>skolemizes to<br>$F(a) \Rightarrow G(b)$<br><br>$\forall x \forall y \exists z\, F(x,y,z) \Rightarrow G(x,y,z)$<br>skolemizes to<br>$\forall x \forall y\, F(x,y,f(x,y)) \Rightarrow G(x,y,f(x,y))$ | **Additional Notes**<br><br>If there are only existential quantifiers, the variables are turned into constants and existential quantifiers dropped.<br><br>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). | **State** – Conjunction of fluents that are ground. |
| **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**<br>1. **Fluents containing variables**. Example: $At(x, y)$<br>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**<br>1. **Action Name and Input Variables**<br>2. **Precondition(s) if any**<br>3. **Effect(s)** | **Action Name and Input Variables**<br>**Name** of the action performed and **any input variables**.<br><br>**Example:** $Fly(p, from, to)$<br>**Action Name:** $Fly$<br>**Variables:** $p, from, to$ | **Preconditions**<br>Aspects of the state that must be true before an action can be performed.<br><br>**Example:**<br>$At(p, from) \land Plane(p)$<br>$\land Airport(from)$<br>$\land Airport(to)$ | **Effects**<br>Action results. Changes in state.<br><br>**Example:**<br>$\neg At(p, From) \land At(p, To)$ | **Complete Example**<br><br>$\boldsymbol{Action}(Fly(p, from, to),$<br><br>$\boldsymbol{Precond}: At(p, from) \land Plane(p)$<br>$\land Airport(from)$<br>$\land Airport(to)$<br><br>$Effect: \neg At(p, From) \land At(p, To))$ |

| | | |
|---|---|---|
| **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 $JFK$.\ | |