Artificial Intelligence I: Exam Preparation Notes

Tomáš Jelínek

January 18, 2025

Contents

1 Disclamer

These exam notes were written mostly using Gemini artificial intelligence based on the lectures.

2 Intelligent Agents, Environment, and Structure of Agents

2.1 Agents and Rational Agents

Agent: An agent is any entity that perceives its environment through **sensors** and acts upon that environment through **actuators**. The agent processes the received information (percepts) and selects actions to influence the environment.

Rational Agent: A rational agent is one that selects actions expected to **maximize its performance measure**, given the evidence provided by the percept sequence and its built-in knowledge. Rationality focuses on ideal performance, not necessarily mimicking human behavior.

Agent Function: A formal, mathematical description of an agent's behavior. It maps any given sequence of percepts (V*) to an action (A).

$$f : V^* \rightarrow A$$

where V is the set of percepts and A is the set of actions.

Agent Program: A concrete implementation of the agent function. It takes the current percept as input and returns an action. The program must be finite and executable on the agent's architecture.

Performance Measure: A function that evaluates the desirability of any given sequence of environment states. It quantifies the agent's success. Defined by the agent's designer, not the agent itself. Should describe the desired properties of the environment, not prescribe specific actions.

- Example: Vacuum cleaner performance measure could be "amount of dirt collected" (bad) or "cleanliness of the floor" (better).

Difference between Rational and Omniscient Agents:

- **Rational agents** maximize *expected* performance based on available information. They may make mistakes due to incomplete or incorrect knowledge.
- **Omniscient agents** know the actual outcome of their actions and maximize *actual* performance. They never make mistakes.

Autonomy: A rational agent should be autonomous. It should learn from experience to compensate for partial or incorrect prior knowledge.

2.2 Properties of Environment

The environment, together with the agent's sensors, actuators, and performance measure, constitutes the **task environment**.

1. **Fully Observable vs. Partially Observable**:
   - **Fully Observable**: The agent's sensors have access to the complete state of the environment at each point in time. (e.g., chess)
   - **Partially Observable**: The agent's sensors only provide partial information about the environment's state. (e.g., autonomous driving)
2. **Deterministic vs. Stochastic**:
   - **Deterministic**: The next state of the environment is completely determined by the current state and the action executed by the agent.
   - **Stochastic**: The next state is determined probabilistically. There's uncertainty about the outcome of actions.
   - **Strategic**: The environment can be modified by other agents. The outcome of an agent's action depends on the actions of other agents. (e.g. chess)
3. **Episodic vs. Sequential**:
   - **Episodic**: The agent's experience is divided into atomic episodes. Each episode consists of the agent perceiving and then acting. The next episode is independent of previous episodes. (e.g., medical diagnosis, image classification of fires on snapshots)
   - **Sequential**: The current action may affect future actions. Decisions are interdependent. (e.g., chess, autonomous driving)
4. **Static vs. Dynamic**:

- **Static**: The environment does not change while the agent is deliberating.
- **Dynamic**: The environment can change while the agent is deliberating.
- **Semidynamic**: The environment itself doesn't change, but the agent's performance score does. (e.g., chess with a clock)

5. **Discrete vs. Continuous**:
   - **Discrete**: The environment, actions, and percepts can be represented with discrete values. (e.g., chess)
   - **Continuous**: The environment, actions, and percepts involve continuous values. (e.g., autonomous driving)
6. **Single Agent vs. Multi-agent**:
   - **Single Agent**: Only one agent is operating in the environment.
   - **Multi-agent**: Multiple agents are operating in the environment. They can be cooperative or competitive.
   - **How to decide if entity is an agent**: Check if it is maximizing its own performance measure.
   - **Example**: Two rovers on Mars, far from each other, not communicating - each is in its own single-agent environment.

Simplest Environment: Fully observable, deterministic, episodic, static, discrete, single agent. **Most Challenging Environment**: Partially observable, stochastic, sequential, dynamic, continuous, multi-agent. (Real-life)

2.3 Typical Agent Structures

Agent = Architecture + Program

- **Architecture**: The physical machinery (computing device, sensors, actuators) on which the agent operates.
- **Program**: Implements the agent function, mapping percepts to actions.

1. **Simple Reflex Agents**:
   - Act based solely on the **current percept**, ignoring the rest of the percept history.
   - Implemented using **condition-action rules** (if-then rules).
   - Suitable for fully observable environments.
   - May get stuck in infinite loops in partially observable environments.
   - Example: Vacuum cleaner that only reacts to the current room's state (clean/dirty).
   - **Code example**:

```
function SIMPLE-REFLEX-AGENT(percept)
    persistent: rules, a set of condition-action rules
    state ← INTERPRET-INPUT(percept)
    rule ← RULE-MATCH(state, rules)
    action ← rule.ACTION
    return action
end function
```

Algorithm 1: Simple Reflex Agent

2. **Model-based Reflex Agents**:

    - Handle partial observability by maintaining an **internal state** that represents the agent's belief about the world.
    - Use a **model of the world**, which includes:
        - **Transition model**: How the world evolves over time, independently of the agent.
        - **Sensor model**: How the agent's percepts relate to the actual state of the world.
    - Update the internal state based on the percept history, transition model, and sensor model.
    - Act based on the **inferred current state**.
    - Still reflex agents, as they react to the current state, not considering future consequences.
    - **Code example**:

```
function MODEL-BASED-REFLEX-AGENT(percept)
    persistent:
        state, the agent's current conception of the world state
        transition_model, a description of how the next state depends on current state and action
        sensor_model, a description of how the current world state is reflected in the agent's percepts
        rules, a set of condition-action rules
        action, the most recent action, initially none
    state ← UPDATE-STATE(state, action, percept, transition_model, sensor_model)
    rule ← RULE-MATCH(state, rules)
    action ← rule.ACTION
    return action
end function
```

Algorithm 2: Model-Based Reflex Agent

3. **Goal-based Agents**:

    - Consider **future consequences** of actions.
    - Have a **goal** or a set of goals representing desirable states.
    - Use **search** and **planning** to find action sequences that achieve the goal.
    - More flexible than reflex agents but can be less efficient.

4. **Utility-based Agents**:

    - Generalization of goal-based agents.
    - Use a **utility function** to assign a value (utility) to each state, representing the desirability of that state.
    - Internalize the performance measure.
    - Can handle conflicting goals and situations with varying probabilities of success.
    - Choose actions that maximize **expected utility**.

2.4 Possible Representations of States

1. **Atomic Representation**:
    - Each state is treated as a **black box**, indivisible, with no internal structure.
    - Used in search algorithms, game-playing, and Markov Decision Processes.
    - Example: State is represented by a unique number or identifier.
2. **Factored Representation**:
    - Each state is represented by a vector of **attributes** or **variables**, each with a specific value.
    - Used in constraint satisfaction, propositional logic, and planning.
    - Example: State in a planning problem represented by a set of propositions (e.g., "block A is on block B" - true/false).
3. **Structured Representation**:
    - States are represented using **objects**, their **attributes**, and **relationships** between objects.
    - Used in first-order logic and knowledge representation.
    - Example: Representing a scene with objects like "car," "person," "road," with attributes like color and speed, and relationships like "car is on road."

2.5 General Learning Agent

- Can improve its performance over time by learning from experience.
- Can operate in initially unknown environments and become more competent than its initial knowledge allows.
- Components:
    - **Performance element**: The part of the agent responsible for selecting actions (can be any of the agent structures described above).
    - **Learning element**: Responsible for making improvements to the performance element.
    - **Critic**: Provides feedback on the agent's performance based on a fixed performance standard. Uses special sensors to evaluate how the agent is doing. (e.g. pain or pleasure sensors)
    - **Problem generator**: Suggests exploratory actions that may lead to new and informative experiences. Balances exploitation (using known good actions) and exploration (trying new actions).

What can be learned: Any part of the agent can be improved through learning, including:

- The mapping from percepts to actions.
- The transition model.
- The sensor model.
- The utility function.

Example: AlphaGo used self-play and a problem generator to discover new moves and improve its performance beyond what it could learn from observing human players.

3 Problem Solving

3.1 Goal-Based Agents

- **Simple Reflex Agent:** Maps percepts directly to actions based on predefined rules.
- **Model-Based Reflex Agent:** Combines current percept with internal state (history) to select actions. Still no planning involved.
- **Goal-Based Agent:** Plans a sequence of actions to achieve a specific goal.
  - **Problem-Solving Agent:** A specific type of goal-based agent that focuses on finding a solution to a well-defined problem.

3.2 Problem Formulation

A well-defined problem consists of:

1. **Initial State:** The starting point of the problem.
2. **Actions:** A set of possible actions available in each state.
3. **Transition Model:** Describes the outcome of each action in each state. Often defined implicitly using a **successor function**.
   - **Successor Function:** Takes a state as input and returns a set of (action, next-state) pairs.
4. **State Space:** The set of all states reachable from the initial state by any sequence of actions.
5. **Path:** A sequence of states connected by actions.
6. **Goal Test:** A function that determines if a given state is a goal state.
7. **Path Cost:** A function that assigns a numeric cost to each path. Often the sum of the costs of individual actions along the path.

3.3 Solution

- **Solution:** A sequence of actions that leads from the initial state to a goal state.
- **Optimal Solution:** A solution with the lowest path cost among all possible solutions.

3.4 Example: Romania Trip

- **Goal:** Reach Bucharest from Arad.
- **States:** Major cities in Romania.
- **Actions:** Driving from one city to another.
- **Path Cost:** Distance between cities.
- **Abstraction:** Simplifying the real world by ignoring details like weather, traffic, exact location within a city, etc.
  - **Valid Abstraction:** An abstract solution can be translated into a solution in the more detailed world.
  - **Useful Abstraction:** Solving the problem at the abstract level is easier than at the detailed level.

3.5 Problem Solving Steps

1. **Goal Formulation:** Define the desired states of the world.
2. **Problem Formulation:** Define states, actions, transition model, goal test, and path cost.
3. **Problem Solving (Search):** Find the best sequence of actions (or states) to reach the goal.
4. **Solution Execution:** Execute the actions in the found solution using a simple reflex agent.

```
Data: percept
Result: action
static: seq (action sequence, initially empty), state (current world state), goal
  (initially null), problem (problem formulation)
state ← UPDATE-STATE(state, percept)
if seq is empty then
    goal ← FORMULATE-GOAL(state) problem ← FORMULATE-PROBLEM(state,
      goal) seq ← SEARCH(problem)
end
action ← FIRST(seq) seq ← REST(seq) return action
```

Algorithm 3: SIMPLE-PROBLEM-SOLVING-AGENT

3.6 Toy Problems

Used to illustrate and compare search techniques:

- **Vacuum World:** States represent agent's location and dirt locations. Actions are Left, Right, Suck.
- **8-Puzzle:** States represent positions of numbered tiles. Actions are moving tiles (Left, Right, Up, Down).
- **8-Queens:** Place 8 queens on a chessboard such that no two queens attack each other.
  - **Incremental Formulation:** States represent partial placements of queens. Actions add a queen.
  - **Complete-State Formulation:** States represent complete placements of queens. Actions move a queen.

3.7 Real-World Problems

- **Route-Finding:** Finding a path from a start location to a goal location.
- **Touring Problems:** Visiting a set of locations (e.g., Traveling Salesman Problem).
- **Product Assembly:** Planning the movements of a robotic arm to assemble a product.

4 Uninformed Search

4.1 State Space Search

- Start at the initial state (root node).
- Check if the current state is a goal state.
- If not, **expand** the state: generate its successor states.

- Select the next state to expand based on a **search strategy**.
- Repeat until a goal state is found or no more states can be expanded.

4.2 Tree Search vs. Graph Search

- **Tree Search:**
  - Explores the state space by generating a search tree.
  - May re-expand the same state multiple times if it's reached through different paths.
  - Simpler to implement.
  - Can be inefficient in terms of time and memory if the state space has many redundant paths.
- **Graph Search:**
  - Avoids re-expanding the same state by keeping track of explored states in a **closed list** (also called **explored set**).
  - Requires more memory to store the closed list.
  - More efficient in terms of time, especially when the state space has many redundant paths.

```
Data: problem, fringe
Result: solution or failure
fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
repeat
    node ← REMOVE-FRONT(fringe) if GOAL-TEST[problem](STATE[node]) then
        return SOLUTION(node)
    end
    fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
until fringe is empty;
return failure
```

Algorithm 4: TREE-SEARCH

```
Data: node, problem
Result: set of nodes
successors ← an empty set
for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
    s ← a new NODE PARENT-NODE[s] ← node ACTION[s] ← action STATE[s] ←
     result PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
     DEPTH[s] ← DEPTH[node] + 1 add s to successors
end
return successors
```

Algorithm 5: EXPAND

```
Data: problem, fringe
Result: solution or failure
closed ← an empty set fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]),
  fringe)
repeat
    node ← REMOVE-FRONT(fringe) if GOAL-TEST[problem](STATE[node]) then
        return SOLUTION(node)
    end
    if STATE[node] is not in closed then
        add STATE[node] to closed fringe ← INSERT-ALL(EXPAND(node, problem),
          fringe)
    end
until fringe is empty;
return failure
```

Algorithm 6: GRAPH-SEARCH

4.3 Node vs. State

- **State:** A representation of the world at a particular point in time.
- **Node:** A data structure used in the search tree. Each node contains:
  - State
  - Parent node
  - Action that led to this node from the parent
  - Path cost (g(n)) from the root to this node
  - Depth of the node

4.4 Fringe (Frontier)

- The set of nodes that have been generated but not yet expanded.
- Also known as the **open list**.
- Implemented as a queue.
- Operations on the fringe:
  - MAKE-QUEUE(element, ...)
  - EMPTY?(queue)
  - FIRST(queue)
  - REMOVE-FIRST(queue)
  - INSERT(element, queue)
  - INSERT-ALL(elements, queue)

4.5 Measuring Performance

- **Completeness:** Does the algorithm guarantee finding a solution if one exists?
- **Optimality:** Does the algorithm find the optimal solution?
- **Time Complexity:** How long does it take to find a solution?
- **Space Complexity:** How much memory is needed to perform the search?

Factors influencing complexity:

- **Branching Factor (b):** Maximum number of successors of any node.
- **Depth (d):** Depth of the shallowest goal node.
- **Maximum Path Length (m):** Maximum length of any path in the state space.
- **Search Cost:** Time and space used by the search algorithm.
- **Total Cost:** Search cost + path cost of the found solution.

4.6 Uninformed Search Strategies

4.6.1 Breadth-First Search (BFS)

- **Strategy:** Expand the shallowest unexpanded node first.
- **Implementation:** Fringe is a FIFO queue.
- **Completeness:** Yes (if b is finite).
- **Optimality:** Yes, if path cost is a non-decreasing function of depth (e.g., all actions have the same cost).
- **Time Complexity:** $O(b^{d+1})$
- **Space Complexity:** $O(b^{d+1})$
- **Issue:** Memory requirements can be very high.

4.6.2 Uniform-Cost Search

- **Strategy:** Expand the node with the lowest path cost g(n) first.
- **Implementation:** Fringe is a priority queue ordered by g(n).
- **Completeness:** Yes, if the cost of each step is greater than or equal to some small positive constant ϵ.
- **Optimality:** Yes.
- **Time Complexity:** $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, where C^* is the cost of the optimal solution.
- **Space Complexity:** $O(b^{1+\lfloor C^*/\epsilon \rfloor})$
- **Note:** This is essentially Dijkstra's algorithm.
- **Issue:** Can be much worse than BFS if there are many small-cost steps.

4.6.3 Depth-First Search (DFS)

- **Strategy:** Expand the deepest unexpanded node first.
- **Implementation:** Fringe is a LIFO stack (or implemented recursively).
- **Completeness:** No (can get stuck in infinite branches).
- **Optimality:** No.
- **Time Complexity:** $O(b^m)$
- **Space Complexity:** $O(bm)$ (can be reduced to $O(m)$ with backtracking).
- **Backtracking:** Generate successors one at a time instead of all at once, reducing space complexity.

4.6.4 Depth-Limited Search (DLS)

- **Strategy:** DFS with a predetermined depth limit l. Nodes at depth l are treated as if they have no successors.
- **Completeness:** No (if l ¡ d).
- **Optimality:** No.
- **Time Complexity:** $O(b^l)$
- **Space Complexity:** $O(bl)$
- **Issue:** How to choose the depth limit l?
  - If l is too small, the solution may not be found.
  - If l is too large, many unnecessary nodes may be explored.
  - Can use domain knowledge (e.g., diameter of the state space) to choose l.

```
Data: problem, limit
Result: solution, failure, or cutoff
return RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem,
 limit)
```

Algorithm 7: DEPTH-LIMITED-SEARCH

```
Data: node, problem, limit
Result: solution, failure, or cutoff
cutoff-occurred? ← false
if GOAL-TEST[problem](STATE[node]) then
    return SOLUTION(node)
end
else if DEPTH[node] = limit then
    return cutoff
end
else
    for each successor in EXPAND(node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit) if result = cutoff then
            cutoff-occurred? ← true
        end
        else if result ≠ failure then
            return result
        end
    end
    if cutoff-occurred? then
        return cutoff
    end
    else
        return failure
    end
end
```

Algorithm 8: RECURSIVE-DLS

4.6.5 Iterative Deepening Search (IDS)

- **Strategy:** Repeatedly apply DLS with increasing depth limits (0, 1, 2, ...).
- **Completeness:** Yes (if b is finite).
- **Optimality:** Yes (if step costs are identical).

- **Time Complexity:** $O(b^d)$
- **Space Complexity:** $O(bd)$
- **Advantages:** Combines the benefits of BFS (completeness, optimality) and DFS (low memory usage).
- **Note:** Although it seems wasteful to re-explore nodes, the overhead is often not significant compared to the cost of exploring the deepest level.
- **Preferred uninformed search method when the search space is large and the depth of the solution is unknown.**

```
Data: problem
Result: solution or failure
for $depth = 0\ to\ \infty$ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth) if $result \neq cutoff$ then
        return result
    end
end
```

Algorithm 9: ITERATIVE-DEEPENING-SEARCH

4.7 Advanced Search Strategies

4.7.1 Bidirectional Search

- **Strategy:** Run two simultaneous searches: one forward from the initial state and one backward from the goal state. Stop when the two frontiers intersect.
- **Motivation:** $b^{d/2} + b^{d/2} << b^d$
- **Completeness:** Yes (if BFS is used in both directions).
- **Optimality:** Not guaranteed (may need additional search to find the optimal solution after the frontiers intersect).
- **Time Complexity:** $O(b^{d/2})$
- **Space Complexity:** $O(b^{d/2})$
- **Difficulties:**
  - Defining predecessors for backward search.
  - Representing the goal state for backward search when the goal is defined by a property rather than a specific state.
  - Requires keeping at least one frontier in memory.

4.7.2 Backward Search

- **Challenges:**
  - Defining predecessors of a state.
  - Representing the goal state implicitly when it's defined by a property.
- **Solutions:**
  - For multiple goal states, use a dummy goal state or a meta-state representing a set of goal states.
- **Note:** Backward search is not used as frequently as forward search due to these difficulties.

4.8 Handling Partial Observability and Non-Determinism

4.8.1 Partial Observability (No Sensors)

- **Belief States:** Instead of working with single states, work with sets of states (belief states) representing the possible states the agent could be in.
- **Conformant Problems:** Problems where a solution can be found even with partial observability.
- **Example:** Vacuum world without sensors. A sequence of actions can still guarantee reaching the goal state.

4.8.2 Non-Deterministic Actions

- **AND-OR Search:** Instead of finding a single path, find a tree of paths (contingency plan) that accounts for all possible outcomes of non-deterministic actions.
- **Contingency Problems:** Problems with non-deterministic actions.

4.8.3 Unknown Actions

- **Exploration Problems:** The agent needs to learn the effects of its actions by exploring the environment.

5 Summary Table

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional S
Complete?	YES*	YES*	NO	YES, if $l \geq d$	YES*	YES*
Time	b^{d+1}	$b^{C*/\epsilon}$	b^m	b^l	b^d	$b^{d/2}$
Space	b^{d+1}	$b^{C*/\epsilon}$	bm	bl	bd	$b^{d/2}$
Optimal?	YES*	YES*	NO	NO	YES*	YES*

Table 1: Comparison of Uninformed Search Algorithms

*Complete if b is finite (BFS), optimal if step costs are all identical.
Key:

- b: branching factor
- d: depth of the shallowest solution
- C*: cost of the optimal solution
- *epsilon*: minimal step cost
- m: maximum depth of the search tree
- l: depth limit

6 Evaluation Function and Heuristic Function

- **Evaluation function f(n)**: Estimates the "desirability" of expanding node n. It typically combines the cost to reach the node and the estimated cost from the node to the goal.

- **Heuristic function h(n)**: Estimates the cost of the cheapest path from node n to a goal state. It is a crucial component of the evaluation function.

Examples of heuristic functions:

- **In a pathfinding problem (e.g., finding a route on a map):**
  - **Straight-line distance (Euclidean distance)** between the current location and the goal location.
  - **Manhattan distance**: Sum of the absolute differences of the Cartesian coordinates. Useful when movement is restricted to a grid.
- **In the 8-puzzle problem:**
  - **Number of misplaced tiles**: Counts the tiles that are not in their correct goal positions.
  - **Manhattan distance**: Sum of the distances of each tile from its goal position (number of moves to get the tile to the goal position, assuming no other tiles).

7 Greedy Best-First Search

Greedy best-first search expands the node that is estimated to be closest to the goal based on the heuristic function.

Evaluation function: $f(n) = h(n)$

Properties:

- **Not optimal**: It can get stuck in local optima and miss the best solution. For example, in the Arad-Bucharest problem, it might choose a path that initially seems to lead quickly towards Bucharest but ends up being longer overall.
- **Not complete (in tree-search)**: It might not find a solution even if one exists, especially if the search space is infinite or contains loops without proper repeated state detection.
- **Complete (in graph-search with repeated state detection)**: If repeated states are detected, it will eventually find a solution if one exists within the search space, because it won't revisit the same states.
- **Time complexity**: $O(b^m)$ in the worst case, where b is the branching factor, and m is the maximum depth of the search tree. The actual performance heavily depends on the quality of the heuristic.
- **Space complexity**: $O(b^m)$ as it keeps all generated nodes in memory.

8 Algorithm A*

Algorithm A* combines the cost to reach a node (g(n)) with the estimated cost to the goal (h(n)).

Evaluation function: $f(n) = g(n) + h(n)$

- $g(n)$: The cost of the path from the start node to node n.
- $h(n)$: The estimated cost of the cheapest path from node n to a goal node.

8.1 Admissible and Consistent Heuristics

- **Admissible heuristic**: A heuristic $h(n)$ is admissible if it never overestimates the cost to reach the goal. Formally, $h(n) \leq h*(n)$ for all nodes n, where $h*(n)$ is the true cost of the cheapest path from n to the goal.
- **Consistent (monotonous) heuristic**: A heuristic $h(n)$ is consistent if for every node n and every successor n' of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n'. Formally, $h(n) \leq c(n, a, n') + h(n')$, where $c(n, a, n')$ is the cost of moving from n to n' via action a.
- **Relationship**: Every consistent heuristic is also admissible, but not vice-versa. Consistency is a stronger condition.

8.2 Completeness and Optimality of A*

8.2.1 Tree-Search

Completeness: A* with an admissible heuristic is complete in finite search spaces. It will eventually find a solution if one exists.

Optimality: A* with an admissible heuristic is optimal if implemented using tree search. It is guaranteed to find the optimal solution (the path with the lowest cost).

Proof of optimality (tree-search):

Let G_2 be a suboptimal goal node in the fringe, and let $C*$ be the cost of the optimal solution.

1. $f(G_2) = g(G_2) + h(G_2) = g(G_2) > C*$ because $h(G_2) = 0$ for a goal node and G_2 is suboptimal.

2. Let n be a node on the optimal path that is currently in the fringe. Since h is admissible, we have $h(n) \leq h*(n)$, where $h*(n)$ is the true cost from n to the goal.

3. Then, $f(n) = g(n) + h(n) \leq g(n) + h*(n) = C*$.

4. Combining (1) and (3), we have $f(n) \leq C* < f(G_2)$.

5. Since A* always expands the node with the lowest f-value, it will expand n before G_2. This means A* will always find the optimal goal before any suboptimal goal.

8.2.2 Graph-Search

Completeness: A* with a consistent heuristic is complete in finite search spaces.

Optimality: A* with a consistent heuristic is optimal if implemented using graph search.

Proof of optimality (graph-search):

1. **Monotonicity**: With a consistent heuristic, the f-values along any path are non-decreasing. If n' is a successor of n: $f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$

2. **First expansion is optimal**: When A* expands a node n, it has found the optimal path to that node. Suppose a better path to n existed. In that case, there would be another node n" on that path in the fringe with a lower f-value (due to monotonicity). A* would have expanded n" first, a contradiction.

3. Therefore, when a goal node is selected for expansion, it is guaranteed to be optimal because it has the lowest f-value among all nodes in the fringe and represents the optimal path to that goal.

8.3 Contours in A*

A* expands nodes in concentric contours of increasing f-cost.

- Nodes within a contour have f-values less than or equal to the contour's value.

- With a better heuristic (more informed/accurate), the contours are more focused towards the goal, leading to fewer node expansions.

8.4 Complexity of A*

- **Time Complexity**:
  - The worst-case time complexity can be exponential, $O(b^d)$, where b is the branching factor and d is the depth of the solution.
  - The actual time complexity depends heavily on the heuristic. A good heuristic can dramatically reduce the number of nodes expanded.
  - If —h(n)-h*(n)— ¡= O(log h*(n)), then the time complexity can be sub-exponential.
- **Space Complexity**:
  - A* keeps all generated nodes in memory, so the space complexity is also typically exponential, $O(b^d)$. This is often the limiting factor for A*.

9 Overcoming Memory Issues of A*

9.1 Iterative Deepening A* (IDA*)

- **Idea**: Run iterative deepening search, but instead of using a depth limit, use an f-cost limit.
- **Algorithm**:
  1. Start with an f-cost limit equal to the estimated cost of the start node (h(start)).
  2. Run a depth-first search, expanding only nodes with an f-cost less than or equal to the current limit.
  3. If a goal is found, return the solution.
  4. Otherwise, set the new f-cost limit to the smallest f-cost of any node that exceeded the previous limit.
  5. Repeat steps 2-4 until a solution is found.

```
Data: problem
Result: solution sequence
static: f-limit, the current f-cost limit;
root, a node;
root ← MAKE-NODE(INITIAL-STATE[problem]);
f-limit ← f-cost(root);
repeat
    solution, f-limit ← DFS-CONTOUR(root, f-limit);
until solution is non-null or f-limit = ∞;
return solution;
```

Algorithm 10: IDA*(problem)

```
Data: node, f-limit
Result: solution sequence, new f-cost limit
static: next-f, the f-cost limit for the next contour, initially $\infty$;
if f-cost[node] ¿ f-limit then
    return null, f-cost[node];
end
if GOAL-TEST[problem](STATE[node]) then
    return node, f-limit;
end
foreach node s in SUCCESSORS(node) do
    solution, new-f $\leftarrow$ DFS-CONTOUR(s, f-limit);
    if solution is non-null then
        return solution, f-limit;
    end
    next-f $\leftarrow$ MIN(next-f, new-f);
end
return null, next-f;
```

Algorithm 11: DFS-CONTOUR(node, f-limit)

- **Properties**:
  - **Complete** if the heuristic is admissible (and the search space is finite).
  - **Optimal** if the heuristic is admissible.
  - **Space complexity**: (O(bd)), where b is the branching factor, and d is the depth of the solution. This is a significant improvement over A*.
  - **Time complexity**: Can be higher than A* due to re-expanding nodes in each iteration. However, in practice, the overhead is often not significant because the number of nodes at each level tends to grow exponentially.

9.2 Recursive Best-First Search (RBFS)

- **Idea**: Mimic best-first search using only linear space. It keeps track of the f-value of the best alternative path available from any ancestor of the current node.
- **Algorithm**:
  1. Expand the best node (lowest f-value) as in standard best-first search.
  2. When expanding a node, if a better alternative path exists from an ancestor (based on the tracked f-values), stop exploring the current path.
  3. As the algorithm backtracks, it updates the f-value of each node along the path with the best f-value of its children (effectively remembering the cost of the best-forgotten subtree).

```
Data: problem
Result: solution or failure
return RBFS(problem, MAKE-NODE(INITIAL-STATE[problem]), $\infty$);
```

Algorithm 12: RECURSIVE-BEST-FIRST-SEARCH(problem)

```
Data: problem, node, f_limit
Result: solution or failure, new f-cost limit
if GOAL-TEST[problem](STATE[node]) then
    return node;
end
successors ← EXPAND(node, problem);
if successors is empty then
    return failure, ∞;
end
foreach s in successors do
    f[s] ← max(g(s) + h(s), f[node]);
end
repeat
    best ← the lowest f-value node in successors;
    if f[best] ¿ f_limit then
        return failure, f[best];
    end
    alternative ← the second-lowest f-value among successors;
    result, f[best] ← RBFS(problem, best, min(f_limit, alternative));
until result ≠ failure;
return result;
```

Algorithm 13: RBFS(problem, node, f_limit)

- **Properties**:
  - **Complete** if the heuristic is admissible (and the search space is finite).
  - **Optimal** if the heuristic is admissible.
  - **Space complexity**: (O(bd)).
  - **Time complexity**: Can be higher than A* due to re-expanding nodes. It is susceptible to excessive node regeneration if the heuristic values change frequently along the path.

9.3 Simplified Memory-Bounded A* (SMA*)

- **Idea**: Utilize all available memory. It expands nodes like A* until memory is full. When memory is full, it drops the worst leaf node (highest f-value) to make space for new nodes.
- **Algorithm**:
  1. Proceed like A*, expanding the best leaf node until memory is full.
  2. When memory is full, drop the worst leaf node (the one with the highest f-value). If there are multiple nodes with the highest f-value, delete the oldest one (the shallowest in the search tree).
  3. Before dropping a node, back up its f-value to its parent. This way, the parent knows the quality of the best path in the forgotten subtree.
  4. Continue expanding and dropping nodes as needed.

```
Data: problem
Result: solution sequence
static: Queue, a queue of nodes ordered by f-cost;
Queue ← MAKE-QUEUE({MAKE-NODE(INITIAL-STATE[problem])});
repeat
    if Queue is empty then
        return failure;
    end
    n ← deepest least-f-cost node in Queue;
    if GOAL-TEST(n) then
        return success;
    end
    s ← NEXT-SUCCESSOR(n);
    if s is not a goal and is at maximum depth then
        f(s) ← ∞;
    end
    else
        f(s) ← MAX(f(n), g(s)+h(s));
    end
    if all of n's successors have been generated then
        update n's f-cost and those of its ancestors if necessary;
    end
    if SUCCESSORS(n) all in memory then
        remove n from Queue;
    end
    if memory is full then
        delete shallowest, highest-f-cost node in Queue;
        remove it from its parent's successor list;
        insert its parent on Queue if necessary;
    end
    insert s on Queue;
until Queue is empty;
```

- **Properties**:
  - **Complete** if there is enough memory to store the shallowest solution path.
  - **Optimal** if there is enough memory to store the shallowest optimal solution path.
  - **Utilizes all available memory**.
  - **Can avoid some node regeneration** compared to IDA* and RBFS, but it can still happen.

10 Weighted A*

Idea: Multiply the heuristic function by a weight (W ¿ 1). This makes the search more greedy, focusing more on quickly reaching the goal and less on finding the absolutely optimal path.

Evaluation function: ($f(n) = g(n) + W \cdot h(n)$)

Properties:

- **Suboptimal**: The solution found may not be optimal, but its cost is guaranteed to be within a factor of W of the optimal solution's cost.
- **Faster**: Typically expands fewer nodes than A*, especially when W is significantly greater than 1.
- **Useful when**:

  - Finding an optimal solution is too time-consuming.
  - A good but not necessarily optimal solution is acceptable.

11 Obtaining Heuristics

11.1 Relaxation

- **Idea**: Solve a relaxed (simplified) version of the problem, where some constraints have been removed. The cost of the optimal solution to the relaxed problem is an admissible heuristic for the original problem.
- **Why it works**: Removing constraints can only make the problem easier to solve, so the cost of the solution to the relaxed problem cannot be greater than the cost of the solution to the original problem.
- **Example (8-puzzle):**
  - **Original problem**: A tile can move from square A to square B if A is adjacent to B and B is blank.
  - **Relaxed problem 1**: A tile can move from square A to square B if A is adjacent to B (ignore whether B is blank). This leads to the Manhattan distance heuristic.
  - **Relaxed problem 2**: A tile can move from square A to square B (ignore adjacency and whether B is blank). This leads to the misplaced tiles heuristic.

11.2 Pattern Databases

- **Idea**: Store the exact costs of solving subproblems (patterns) in a database. For example, in the 8-puzzle, a pattern might be the configuration of a subset of tiles.
- **Algorithm**:
  1. Choose a set of patterns (e.g., the positions of tiles 1-4 in the 8-puzzle).
  2. For each possible pattern, solve the subproblem of getting those tiles into their correct goal positions using a backward search from the goal.
  3. Store the cost of solving each pattern in the database.
  4. When evaluating a node, look up the costs of the patterns that match the current state. The heuristic value can be the maximum cost of all matching patterns or, if the patterns are disjoint, the sum of their costs.
- When using multiple pattern databases, you can take the maximum value among them as heuristic.

Example (8-puzzle):

- Create a database for the positions of tiles 1-4.
- Create another database for the positions of tiles 5-8.
- When evaluating a node, look up the costs in both databases and take the maximum.

Properties:

- **Admissible** if constructed correctly (e.g., using disjoint patterns).
- **Can be very effective**: Pattern databases can provide much more accurate heuristics than simple heuristics like Manhattan distance.
- **Memory intensive**: Storing the pattern database can require a lot of memory, especially for complex problems.

12 Dominance

- If $h_2(n) => h_1(n)$ for all n, then h_2 dominates h_1.
- A* with h_2 will never expand more nodes than A* with h_1.
- It is generally better to use a heuristic that dominates others, as long as it is still admissible and computationally feasible.

13 Example: Applying Heuristic Search to the 8-Puzzle

Let's illustrate how to apply some of the discussed concepts to the 8-puzzle problem.

13.1 Problem Definition

The 8-puzzle is a sliding puzzle that consists of a 3x3 grid with 8 numbered tiles and one empty space. The goal is to rearrange the tiles by sliding them into the empty space until they reach a specific goal configuration.

State: A particular arrangement of the 8 tiles and the blank space within the 3x3 grid.

Actions: The possible actions are:

- Move the blank space up.
- Move the blank space down.
- Move the blank space left.
- Move the blank space right.

Each action has a cost of 1.

Goal State: A specific target configuration of the tiles, usually:

1 2 3 8 4 7 6 5

13.2 Heuristics

1. **Misplaced Tiles**:
   - $h_1(n)$ = the number of tiles that are not in their goal positions.
   - Admissible because each misplaced tile needs at least one move to reach its correct position.
2. **Manhattan Distance**:
   - $h_2(n)$ = the sum of the Manhattan distances of each tile from its goal position.
   - Admissible because each tile needs at least a certain number of moves (its Manhattan distance) to reach its goal position, regardless of other tiles.
   - h_2 dominates h_1.

13.3 Example Search (Illustrative)

Let's consider a simplified example using A* with the Manhattan distance heuristic.

Start State:

Use code with caution.

2 8 3 1 6 4 7 5

Goal State:

Use code with caution.

1 2 3 8 4 7 6 5

Heuristic Calculation (Manhattan Distance):

Tile	Start Position	Goal Position	Manhattan Distance
1	(2,1)	(1,1)	1
2	(1,1)	(1,2)	1
3	(1,3)	(1,3)	0
4	(2,3)	(2,3)	0
5	(3,3)	(3,3)	0
6	(2,2)	(3,2)	1
7	(3,1)	(3,1)	0
8	(1,2)	(2,1)	2
Total			**5**

$h(start) = 5$
$g(start) = 0$ (since it's the start node)
$f(start) = g(start) + h(start) = 0 + 5 = 5$
A* would then:

1. Expand the start node, generating its possible successor states.
2. Calculate the f-value for each successor (g(n) + h(n)).
3. Choose the successor with the lowest f-value to expand next.
4. Continue this process until a goal state is selected for expansion.

Note: This is a simplified illustration. A full A* search would involve exploring multiple paths and potentially updating f-values as better paths are found.

13.4 Relaxed Problem Heuristics

As discussed earlier, we can derive admissible heuristics by solving relaxed versions of the 8-puzzle:

1. **Relaxation 1**: A tile can move from A to B if A is adjacent to B.
    - Leads to the **Manhattan distance** heuristic.
2. **Relaxation 2**: A tile can move from A to B if B is blank.
3. **Relaxation 3**: A tile can move from A to B.
    - Leads to the **misplaced tiles** heuristic.

13.5 Pattern Database Heuristic

We could create a pattern database for the 8-puzzle by considering subsets of tiles (e.g., tiles 1-4).
1. **Pattern Definition:** For example, a pattern could be defined by the positions of tiles 1, 2, 3, and 4, ignoring the other tiles. 2. **Database Creation:**

- Perform a backward search from the goal state, considering only the tiles in the pattern.
- For each reachable pattern, store the minimum number of moves required to get those tiles into their correct positions.

3. **Heuristic Calculation:**

- When evaluating a state, identify the pattern formed by the relevant tiles.
- Look up the cost of that pattern in the database.
- This cost is an admissible heuristic for the number of moves needed to solve the subproblem defined by the pattern.

Example:
If a pattern database for tiles 1-4 contains the following entry:
Use code with caution.
Pattern: 2 1 3 (positions of tiles 1, 2, 3, 4) 4 Cost: 5
This means that any state where tiles 1, 2, 3, and 4 are in the configuration shown above requires at least 5 moves to get those tiles into their goal positions.

14 Conclusion

Informed search algorithms, particularly A* and its variants, are powerful tools for solving problems where a heuristic function can guide the search towards the goal. Understanding the concepts of admissible and consistent heuristics, as well as techniques like relaxation and pattern databases, is crucial for designing effective heuristic search strategies.

Remember that the choice of algorithm and heuristic depends on the specific problem and the trade-offs between solution quality, time complexity, and space complexity. For example, if memory is a major constraint, IDA* or RBFS might be preferred over A*. If a quick, reasonably good solution is acceptable, Weighted A* could be a good choice. And if a very accurate heuristic is needed, pattern databases might be worth the extra memory cost.

15 Local Search

15.1 Concept of Local Search

Local search algorithms are used for optimization problems where the goal is to find the best solution among a set of possible solutions, and the path to the solution is not important. Unlike systematic search algorithms, local search focuses on exploring a small part of the search space around the current solution. Key aspects:

- **State Representation:** Only maintains a single current state (as opposed to a sequence of states), often in constant memory.
- **State Modification:** Iteratively modify the current state to explore the search space.
- **Objective Function:** An objective function defines the "quality" of a given state, allowing the algorithms to seek improvements. This is a crucial part that differs from the search algorithms we've seen before.
- **Path Ignorance:** Local search algorithms generally do not store or care about the path taken to reach the current state.
- **Optimization Goal:** Local search aims at finding the optimal state, which is where the objective function is at it's maximum (or minimum).
- **Landscape**: The search space can be visualized as a "landscape", where the states are coordinates and the objective function gives the elevation.

15.2 Hill-Climbing

15.2.1 Basic Hill-Climbing

Hill-climbing, also known as gradient descent if maximizing or gradient ascent if minimizing, is a simple local search technique that moves towards states that improve the objective function. It only considers the neighboring states and goes to the best one.

- **Greedy Approach:** Selects the best neighbor state without considering future states.
- **Neighborhood:** Explores the states that are one step away from the current one.

```
Input: problem
Output: a state that is a local maximum
current ← problem.INITIAL ;
while true do
    neighbor ← highest-valued successor state of current;
    if VALUE(neighbor) < VALUE(current) then
        return current;
    end
    current ← neighbor;
end
```

Algorithm 14: Hill-Climbing

Problems with Hill-Climbing:

- **Local Optima:** The search can get stuck in a local optimum where neighbors are not better (as opposed to the global optimum where every other state is worse).
- **Ridges:** A sequence of local optima can be difficult to navigate.
- **Plateaux:** A flat area of the state-space landscape (where progress is zero) can be challenging; Hill climbing can cycle in these plateaux.

15.2.2 Stochastic Hill-Climbing

Chooses from the uphill moves randomly. The probability of choosing the move may depend on the steepness of the uphill move (higher for a steeper move)

- It's slower but can find better solutions.

15.2.3 First-Choice Hill-Climbing

Stochastic hill-climbing is performed until the current state improves, after which it chooses that move.

- Can be useful for when the number of successors is very big.

15.2.4 Random-Restart Hill-Climbing

Starts multiple hill-climbing searches from random initial states and chooses the best solution across all runs.

- Can escape local optima, which can be useful in different situations.
- If the probability of the hill-climbing to succeed is p, then the expected number of restarts is $\frac{1}{p}$.

15.3 Simulated Annealing

Simulated annealing balances hill-climbing with random moves, inspired by the annealing process in metallurgy.

- **Temperature:** A 'temperature' variable is gradually reduced (cooling scheme).
- **Probabilistic Acceptance:** Moves to worse states with a probability based on the temperature; allows escaping local minima (this probability is reduced as the temperature is lowered).
- **Global Optimization:** It's more likely to find the global optimum compared to basic hill-climbing.

```
Input: problem, schedule
Output: a solution state
$current \leftarrow problem.INITIAL$ ;
for $t \leftarrow 1$ to $\infty$ do
    $T \leftarrow schedule(t)$ ;
    if $T = 0$ then
        return $current$ ;
    end
    $next \leftarrow$ a randomly selected successor of $current$;
    $\Delta E \leftarrow$ VALUE($current$) − VALUE($next$);
    if $\Delta E > 0$ then
        $current \leftarrow next$ ;
    end
    else
        $current \leftarrow next$ only with probability $e^{\Delta E/T}$ ;
    end
end
```

Algorithm 15: Simulated Annealing

15.4 Population-Based Techniques

These techniques explore the search space using a set of states.

15.4.1 Local Beam Search

Keeps track of k states simultaneously.

- **Parallel Exploration:** Generates all successors of all k states at each step.
- **Selection:** Selects the k best successors, repeating the process.
- **Information Sharing:** Useful information is passed among parallel searches.
- **Resource Management:** Allocates resources to more promising search threads.
- **Lack of Diversity:** Can suffer from lack of diversity, which can lead it to local optima.

15.4.2 Stochastic Beam Search

Same as local beam search but the next states are randomly selected based on the state's objective value.

15.4.3 Genetic Algorithms (GAs)

GAs simulate biological evolution to optimize a function.

- **Population Initialization:** Starts with a set of k random states (population).
- **String Representation:** Each state is represented as a string of characters.
- **Fitness Evaluation:** The objective function evaluates the fitness of each state.
- **Selection:** Parents are selected based on their fitness (better fit states have a higher chance of reproduction).
- **Crossover:** Combine parts of parent strings to create new states (offspring).
- **Mutation:** Introduce small changes to offspring strings (random mutations).
- **Natural Selection:** Resembles the process of natural selection.

```
Input: population, fitness function
Output: an individual
repeat
    weights ← WEIGHTED-BY(population, fitness);
    population2 ← empty list;
    for i ← 1 to SIZE(population) do
        parent1, parent2 ← WEIGHTED-RANDOM-CHOICES(population, weights,
          2);
        child ← REPRODUCE(parent1, parent2);
        if small random probability then
            child ← MUTATE(child);
        end
        add child to population2;
    end
    population ← population2;
until some individual is fit enough or enough time has elapsed;
return the best individual in population, according to fitness ;
```

Algorithm 16: Genetic Algorithm

16 On-line Search

16.1 Formulation and Comparison with Offline Search

Offline Search:

- The algorithm computes a complete solution before executing it.
- Assumes full knowledge of the environment.
- Executes the plan without further interaction (no feedback).

On-line Search:

- Interleaves computing and acting, so the algorithm only makes action decision based on its current state.
- The algorithm does not know the results of actions or the overall environment.
- Selects, executes actions, observes, and then computes the next action.
- Useful for dynamic, semidynamic, and non-deterministic environments.
- Helpful when actions or results of actions are unknown.

16.2 Competitive Ratio and Safely Explorable Environments

Competitive Ratio:

- A measure of the quality of online algorithm compared to offline algorithm.
- Calculated as $\frac{\text{cost of online solution}}{\text{cost of optimal offline solution}}$.
- Can be infinite (e.g., in dead-end states where the algorithm would not know of a way to make progress).

Safely Explorable Environments:

- Every reachable state can lead to the goal.
- Necessary to guarantee that the agent can actually achieve its goal.
- No bounded competitive ratio when there are paths of unbounded cost.
- Adversary arguments can extend path arbitrarily
- Online performance is often evaluated in terms of the size of entire state space.

16.3 On-line DFS

An on-line version of depth-first search, where each node can only be expanded when it is physically occupied.

- **Successor Discovery:** Discovers successors when it occupies them.
- **Local Ordering:** Expands nodes in a local order.
- **Memory:** It remembers visited states, which is essential to avoid looping and exploring unnecessarily.
- **Reversible Actions:** Only works in state spaces where actions are reversible.
- **Worst Case:** Every link is traversed twice (forward and backward) in the worst case.

```
**Input:** problem, s'
**Output:** an action
*persistent:* result, a table mapping $(s, a)$ to $s'$, initially empty;
*persistent:* untried, a table mapping $s$ to a list of untried actions;
*persistent:* unbacktracked, a table mapping $s$ to a list of states never backtracked to;
if $problem.IS-GOAL(s')$ then
    return $stop$;
end
if $s'$ is a new state (not in untried) then
    $untried[s'] \leftarrow problem.ACTIONS(s')$;
end
if $s$ is not null then
    $result[s, a] \leftarrow s'$ add $s$ to the front of $unbacktracked[s']$;
end
if $untried[s']$ is empty then
    if $unbacktracked[s']$ is empty then
        return $stop$;
    end
    else
        $a \leftarrow$ an action $b$ s.t. $result[s', b] = POP(unbacktracked[s'])$; $s' \leftarrow null$;
    end
end
else
    $a \leftarrow POP(untried[s'])$;
end
$s \leftarrow s'$;
return $a$;
```

Algorithm 17: Online DFS

16.4 Learning Real-Time A* (LRTA*)

LRTA* performs local steps while learning the outcome of each action and the distance to the goal.

- **Local Step:** Moves to a neighbor state and learns new information.
- **H-function Update:** Adjusts estimates (H) of the distance to the goal.
- **Cost Estimates:** Uses H and the cost of steps to select actions.
- **Action Selection:** Prioritizes unexplored states to make better progress.
- **Exploration with Optimization:** Balances exploration and optimization.

```
**Input:** problem, $s'$, h
**Output:** an action
*persistent:* result, a table mapping $(s, a)$ to $s'$, initially empty;
*persistent:* H, a table mapping $s$ to a cost estimate, initially empty;
if $IS - GOAL(s')$ then
    return $stop$;
end
if $s'$ is a new state (not in H) then
    $H[s'] \leftarrow h(s')$;
end
if $s$ is not null then
    $result[s, a] \leftarrow s'$;
    $H[s] \leftarrow \min_{b \in ACTIONS(s)} LRTA* -COST(s, b, result[s, b], H)$;
end
$a \leftarrow \underset{b \in ACTIONS(s')}{\text{argmin}} LRTA* -COST(problem, s', b, result[s', b], H)$;
$s \leftarrow s'$;
return $a$;
```

Algorithm 18: LRTA* Agent

```
**Input:** problem, s, a, s', H
**Output:** a cost estimate
if $s'$ is undefined then
    return $h(s)$;
end
else
    return $problem.ACTION - COST(s, a, s') + H[s']$;
end
```

Algorithm 19: LRTA* Cost

17 Constraint Satisfaction Problems

17.1 Definition of a Constraint Satisfaction Problem (CSP)

A Constraint Satisfaction Problem (CSP) is defined by:

- A finite set of **variables** $X = \{X_1, X_2, ..., X_n\}$. Each variable represents a component of the problem's state.
- A finite set of **domains** $D = \{D_1, D_2, ..., D_n\}$. Each domain D_i specifies the set of possible values for variable X_i.
- A finite set of **constraints** $C = \{C_1, C_2, ..., C_m\}$. Each constraint C_j specifies a relation over a subset of variables, restricting the values that the variables can simultaneously take.

A **constraint** is a relation over a subset of variables specifying the values of variables that are compatible. Constraints can be specified:

- In extension: explicitly listing all tuples of allowed values.
- Using a formula: for example $X_i \neq X_j$ or $X_i < X_j$.

17.2 Examples of CSPs

1. **Sudoku**:
    - Variables: Each cell in the 9x9 grid.

- Domains: $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ for each cell.
- Constraints:
  - No same digit can appear twice in the same row.
  - No same digit can appear twice in the same column.
  - No same digit can appear twice in the same 3x3 subgrid.

2. **Graph Coloring**:
   - Variables: Each node in the graph.
   - Domains: Set of colors (e.g., {red, blue, green}).
   - Constraints: Adjacent nodes cannot have the same color.

3. **N-Queens Problem**:
   - Variables: Each column where we can place the Queen.
   - Domains: Row numbers where we can place the Queen $\{1, 2, ..., N\}$ (N = Number of queens).
   - Constraints: Queens cannot attack each other - they have to be in the different columns, rows and diagonals.

4. **Scheduling**:
   - Variables: Tasks to be scheduled.
   - Domains: Time slots in which a task can be done.
   - Constraints: Time constraints that specify that specific tasks have to happen before others and that we cannot do two task at the same time.

17.3 Applying Problem-Solving Techniques to CSPs

Backtracking Search: A depth-first search algorithm that explores the space of partial assignments.

- **Initial State**: Empty assignment.
- **Actions**: Assign a value from a variable's domain, while ensuring constraints are not violated.
- **Goal Test**: Check if assignment is complete and consistent.

Why is it appropriate?:

- CSPs can be described by state space defined via assigning values to the variables.
- Depth-first search can efficiently explore such a state space.
- It is a general-purpose algorithm that works for different CSPs.
- It allows us to utilize different variable/value orderings.
- We can combine backtracking with constraint propagation to reduce state space.

```
Input: A CSP problem csp
Output: A solution or failure
return Backtrack(csp, {});

Function Backtrack(csp, assignment):
    if assignment is complete then
        return assignment;
    end
    var ← Select-Unassigned-Variable(csp, assignment);
    foreach value in Order-Domain-Values(csp, var, assignment) do
        if value is consistent with assignment then
            add {var = value} to assignment;
            inferences ← Inference(csp, var, assignment);
            if inferences ≠ failure then
                add inferences to csp;
                result ← Backtrack(csp, assignment);
                if result ≠ failure then
                    return result;
                end
                remove inferences from csp;
            end
            remove {var = value} from assignment;
        end
    end
    return failure;
```

Algorithm 20: Backtracking Search

17.4 Principles of Variable and Value Ordering

17.4.1 Variable Ordering Heuristics

- **Most Restricted Variable (dom heuristic)**: Choose the variable with the smallest remaining domain size.
    - Rationale: If a variable has a small domain it is more likely to fail if wrong value is chosen, so it makes sense to try to find the value for that variable first.
    - Example: In Sudoku, choose the cell with the fewest possible values.
- **Most Constrained Variable (deg heuristic)**: Choose the variable that is involved in the largest number of constraints.
    - Rationale: If a variable is in many constraints, the assignment of value for this variable might reduce the domain space of other variables, so we should try to find a value for that variable first.
    - Example: In graph coloring, choose the node with the most neighbors.
- **Combination (dom+deg heuristic)**: Used when the *dom* heuristic does not choose single variable.
- **Fail-first principle**: All above heuristics are based on this principle and try to select variables with the highest probability of failure first.

17.4.2 Value Ordering Heuristics

- **Succeed-first principle**: Choose the value that is most likely to lead to a solution.
- **Least Restricting Value**: Choose the value for a variable that eliminates the fewest values from the domains of the other, unassigned variables.

    - Rationale: Keeps the flexibility in the problem.
    - Example: In map coloring, choose the color that leaves the most options available to the neighboring countries.

- **Problem-dependent heuristics**:
    - Rationale: Generally, finding the best values is hard and therefore problem specific heuristics are used.
    - Example: Relaxing the problem and finding the solution for a relaxed problem and using those values in the solution of original problem.

17.5 Arc Consistency

17.5.1 Definition

A binary constraint between two variables X_i and X_j is **arc consistent** if for every value x in the domain of X_i, there exists a value y in the domain of X_j such that the constraint is satisfied.

AC-3 (Arc Consistency Algorithm 3) is an algorithm used to enforce arc consistency in a CSP. Arc consistency is a fundamental concept in constraint satisfaction that helps reduce the search space by ensuring that the values in a variable's domain are consistent with the constraints involving that variable and its neighbors.

AC-3 is used for constraint propagation. It aims to:

- Reduce Domains: Eliminate values from the domains of variables that cannot be part of any solution because they violate constraints with neighboring variables.
- Detect Inconsistencies Early: If a variable's domain becomes empty during the process, it means there's no solution to the CSP, and we can stop the search process immediately.

17.5.2 Algorithm AC-3

```
Input: A CSP with variables, domains, and constraints
Output: True if arc consistency is achieved, False if inconsistency is found
queue ← All arcs in the CSP;
while queue is not empty do
    $(X_i, X_j)$ ← POP(queue);
    if REVISE($X_i, X_j$) then
        if domain of $X_i$ is empty then
            return False;
        end
        for each $X_k \in$ NEIGHBORS($X_i$) $\setminus \{X_j\}$ do
            add $(X_k, X_i)$ to queue;
        end
    end
end
return True;

function REVISE($X_i, X_j$)
revised ← False;
for each value $x$ in domain of $X_i$ do
    if no value $y$ in domain of $X_j$ allows $(x, y)$ to satisfy constraint then
        delete $x$ from domain of $X_i$ ;
        revised ← True;
    end
end
return revised;
```

Algorithm 21: Algorithm AC-3

Time complexity: $O(ed^3)$, where e is the number of constraints, and d is the maximum domain size.

17.6 k-Consistency

17.6.1 Definition

A CSP is k-consistent if, given any consistent assignment to any k-1 variables, there exists a consistent value for any k-th variable.

- **Arc consistency (AC)** = 2-consistency.
- **Path consistency (PC)** = 3-consistency.

17.6.2 Relation to Backtrack-Free Search

If a problem is k-consistent for k=1, ..., n, then a solution can be found without backtracking, because we can find a consistent value for each next variable.

Drawback: Time complexity of achieving k-consistency grows exponentially with k.

17.6.3 Example of a Global Constraint

all_different($X_1, ..., X_k$) constraint:

- Requires that all variables X_1 to X_k have different values.
- Can be solved efficiently using maximum matching in bipartite graphs.
- The graph consists of variable nodes on one side and values on the other, with arcs connecting each variable to all its values.

- The algorithm removes values (arcs) that cannot be part of maximal matching.

17.7 Forward Checking and Look Ahead Techniques

17.7.1 Forward Checking

- After assigning a value to the current variable, we check the future constraints (constraints between the current variable and unassigned variables).
- Values that would cause conflicts in the future are removed from the domains of the unassigned variables.
- It can detect inconsistencies before an actual error and prune dead branches.
- Improves efficiency by avoiding fruitless assignments.

17.7.2 Look Ahead Techniques (Constraint Propagation)

- Checks constraints between unassigned variables and propagates effects of assignments through the whole CSP.
- For example it checks for single values and empty domains in unassigned variables and prunes dead branches earlier then in forward checking.
- Implemented through arc consistency.

18 Core Properties of Environment for Adversarial Search

To apply adversarial search, certain characteristics are required:

- **Multi-agent environment**: There are multiple agents, at least two, whose actions impact each other.
- **Competitive environment**: Agents have opposing goals, creating conflict. Often, this is represented as a *zero-sum game*, where one agent's gain is another's loss.
- **Turn-taking**: Agents typically take turns making decisions. This can be deterministic (like chess) or stochastic (like backgammon).
- **Perfect or Imperfect Information**:
  - *Perfect Information*: All agents have complete knowledge of the game state at all times (e.g., chess, tic-tac-toe).
  - *Imperfect Information*: Some information is hidden from agents (e.g., poker, bridge).
- **Well-defined states and transitions**: The game's rules specify states and how actions change the game state.
- **Terminal states**: A game must have clear terminal states, where it is decided who won or a draw occurred.
- **Utility function**: A function that provides a numerical score of the game's outcome for each terminal state.

19 Mini-Max Search

The *mini-max* algorithm is a recursive search algorithm for two-player, turn-based zero-sum games. It assumes both players play optimally. Here's how it works:

- **Core Idea**: The algorithm simulates the best possible actions for each player. It assumes a player tries to *maximize* their utility while the opponent tries to *minimize* it.
- **Algorithm Structure**:
  - The algorithm explores the game tree depth-first, calculating utility values bottom-up.
  - *MAX* nodes: The algorithm selects the action that leads to the maximum utility.
  - *MIN* nodes: The algorithm selects the action that leads to the minimum utility.
- **Pseudocode**:

```
MINIMAX-SEARCH(game, state);
player ← game.To-MOVE(state);
(value, move) ← MAX-VALUE(game, state);
return move;

MAX-VALUE(game, state);
if IS-TERMINAL(state) then
    return UTILITY(state, player), null;
end
v ← −∞;
foreach a in ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a));
    if v2 > v then
        v ← v2;
        move ← a;
    end
end
return v, move;

MIN-VALUE(game, state);
if IS-TERMINAL(state) then
    return UTILITY(state, player), null;
end
v ← +∞;
foreach a in ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a));
    if v2 < v then
        v ← v2;
        move ← a;
    end
end
return v, move;
```

Algorithm 22: Mini-Max Algorithm

- **Time Complexity**: $O(b^m)$, where b is the branching factor and m is the maximum depth.
- **Space Complexity**: $O(bm)$ for depth-first search or $O(b^m)$ when considering full tree.
- **Limitations**: It can be computationally expensive for complex games since it must explore the entire game tree.

20 Alpha-Beta Search

Alpha-beta pruning is an optimization technique for the mini-max algorithm that reduces the number of nodes explored in the game tree. It achieves this by eliminating branches that cannot affect the final result.

- **Core Idea**: Maintain two bounds (α and β) to represent the best possible values for the maximizing and minimizing player respectively along the current path of the search.
  - α: The best (maximum) value that MAX has found so far.
  - β: The best (minimum) value that MIN has found so far.
- **Algorithm Structure**:
  - When exploring *MAX* nodes, if a value is found greater than β, further exploring of a branch can be cut off, because *MIN* player would not chose such branch as there is a better move already found for the *MIN* player at the higher level of the tree.
  - When exploring *MIN* nodes, if a value is found smaller than α, further exploring of a branch can be cut off, because *MAX* player would not chose such branch as there is a better move already found for the *MAX* player at the higher level of the tree.
- **Time Complexity**: Best case $O(b^{m/2})$, assuming perfect move ordering. Worst case $O(b^m)$.
- **Space Complexity**: $O(bm)$.
- **Advantages**: Significantly improves efficiency compared to Mini-Max, allows deeper search within same time.

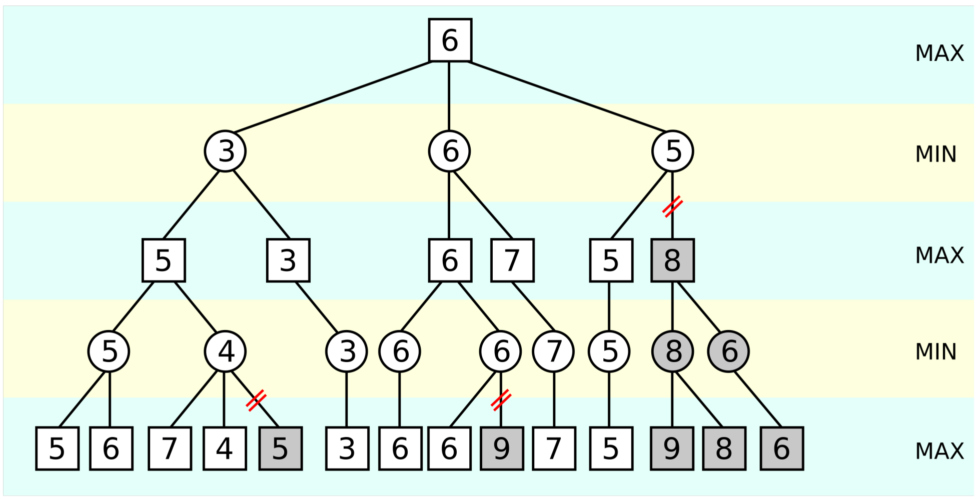


Figure 1: Alpha-Beta Pruning Example

```
ALPHA-BETA-SEARCH(game, state);
player ← game.To-MOVE(state);
(value, move) ← MAX-VALUE(game, state, $-\infty$, $+\infty$);
return move;

MAX-VALUE(game, state, $\alpha$, $\beta$);
if IS-TERMINAL(state) then
    return UTILITY(state, player), null;
end
$v \leftarrow -\infty$;
foreach a in ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), $\alpha$, $\beta$);
    if $v2 > v$ then
        $v \leftarrow v2$;
        move ← a;
        $\alpha \leftarrow \max(\alpha, v)$;
        if $v \geq \beta$ then
            return v, move;
        end
    end
end
return v, move;

MIN-VALUE(game, state, $\alpha$, $\beta$);
if IS-TERMINAL(state) then
    return UTILITY(state, player), null;
end
$v \leftarrow +\infty$;
foreach a in ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), $\alpha$, $\beta$);
    if $v2 < v$ then
        $v \leftarrow v2$;
        move ← a;
        $\beta \leftarrow \min(\beta, v)$;
        if $v \leq \alpha$ then
            return v, move;
        end
    end
end
return v, move;
```

Algorithm 23: Alpha-Beta Algorithm

21 Evaluation Function

An *evaluation function* estimates the utility of a non-terminal state. It's used when a search has to cut off before reaching the terminal states.

- **Purpose**: Provides a heuristic estimate for the quality of a game state without the need for full depth search.
- **Properties**:
  - *Consistency*: Must order terminal states correctly with respect to the true utility function.
  - *Speed*: Must be fast to compute, as it is used on a large number of states.

  - *Correlation*: Non-terminal states should be strongly correlated with the chances of winning.

- **Examples**:
  - *Expected Value:* Based on selected categories of states and their proportion of winning states. For example: EVAL $= (0.72 \times +1) + (0.20 \times -1) + (0.08 \times 0) = 0.52$
  - *Material Value (Chess):* A numerical score based on the value of pieces. $pawn = 1, knight = bishop = 3, rook = 5, queen = 9$. For example: $EVAL(s) = w_1 f_1(s) + w_2 f_2(s) + ... + w_n f_n(s)$

22 Quiescent and Horizon Effect

- **Quiescent Search**:
  - **Problem:** Evaluation functions are unstable in non-quiescent positions, where major actions like taking a piece are expected in the next few moves.
  - **Solution:** Continue the search deeper until a *quiescent* state is reached (the state in which the estimate of state is stable). A few more selected moves (e.g., captures) are explored.
- **Horizon Effect**:
  - **Problem:** An unavoidable bad situation can be delayed after the cut-off limit and is not recognised as bad.
  - **Solution:**
    - *Singular Extensions:* If a move is clearly better, it is explored further.

23 Stochastic Games and Expected Mini-Max

Stochastic games are games that include an element of chance, like the dice rolls in backgammon.

- **Core Idea:** Extend the game tree with *chance nodes* representing possible random events.
- **Expected Mini-Max Value**: Instead of a direct minimax value, we compute an *expected value* at chance nodes, weighted by the probabilities of the chance outcomes:

$$\text{EXPECTIMINIMAX-VALUE}(n) = \begin{cases} \text{UTILITY}(n) & \text{if } n \text{ is a terminal n} \\ \max_{s \in \text{successors}(n)} \text{EXPECTMINIMAX-VALUE}(s) & \text{if MAX plays in } n \\ \min_{s \in \text{successors}(n)} \text{EXPECTMINIMAX-VALUE}(s) & \text{if MIN plays in } n \\ \sum_{s \in \text{successors}(n)} P(s) \cdot \text{EXPECTMINIMAX}(s) & \text{if } n \text{ is a chance nod} \end{cases}$$

  Where P(s) is the probability of reaching succesor state s.

- **Limitations**:
  - **Time Complexity**: $O(b^m n^m)$, where n is the number of random moves.
  - **Evaluation function** absolute values can play a role since now we have a weighted average. The evaluation function should be a linear transformation of expected utility.
  - **Cut-offs**: Can be used as long as the evaluation function is bounded and the expected value can be bounded.

24 Monte Carlo Tree Search (MCTS)

MCTS is a simulation-based search algorithm that balances exploration and exploitation in the search process. MCTS is used to explore deep but narrow parts of game tree.

- **Core Idea**: Instead of using a evaluation function, use the average utility over a number of simulations of complete games.
- **Main Steps**:
  - **Selection**: From the root of the tree, choose a move by traversing the tree using the *selection policy* (UCT) until a leaf node is reached.
  - **Expansion**: Generate a new child node from the leaf.
  - **Simulation**: Starting from the new child, simulate a full game using the *playout policy* (often random).
  - **Back-propagation**: Update the utility of each node on the path from the expanded node back to the root by the simulation results.
- **Exploration vs Exploitation**:
  - *Exploration*: Discover unexplored parts of the tree.
  - *Exploitation*: Focus on the parts of the tree that appear promising.
  - *UCT (Upper Confidence Bound Applied to Trees)*:

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log(N(\text{parent}(n)))}{N(n)}}$$

    Where:
    * $\frac{U(n)}{N(n)}$ is the exploitation term (average utility of node n),
    * $\sqrt{\frac{\log(N(\text{parent}(n)))}{N(n)}}$ is the exploration term (prefers less explored nodes), and
    * C is a constant balancing exploration and exploitation. Typically set to $\sqrt{2}$

25 Conclusion

Adversarial search provides powerful techniques for agents to make decisions in competitive, multi-agent scenarios. It is also applicable to stochastic games and when perfect information is not available. MCTS is a popular alternative to alpha-beta when there is a good playout policy.

26 Knowledge Representation and Propositional Logic

26.1 Knowledge-Based Agents

- **Definition:** A knowledge-based agent is an intelligent agent that uses knowledge about the world to make decisions. It maintains an internal representation of the world, called a knowledge base (KB), which consists of sentences expressed in a formal language (e.g., propositional logic).
- **Components:**
  - **Knowledge Base (KB):** A set of sentences that represent facts and rules about the world.

  - **Inference Mechanism:** A process for deriving new sentences from the existing ones. The process of deriving conclusions, and use that to deduce what to do.
  - **Percepts:** The agent gathers these from the environment.
  - **Actions:** The agent performs in the environment.

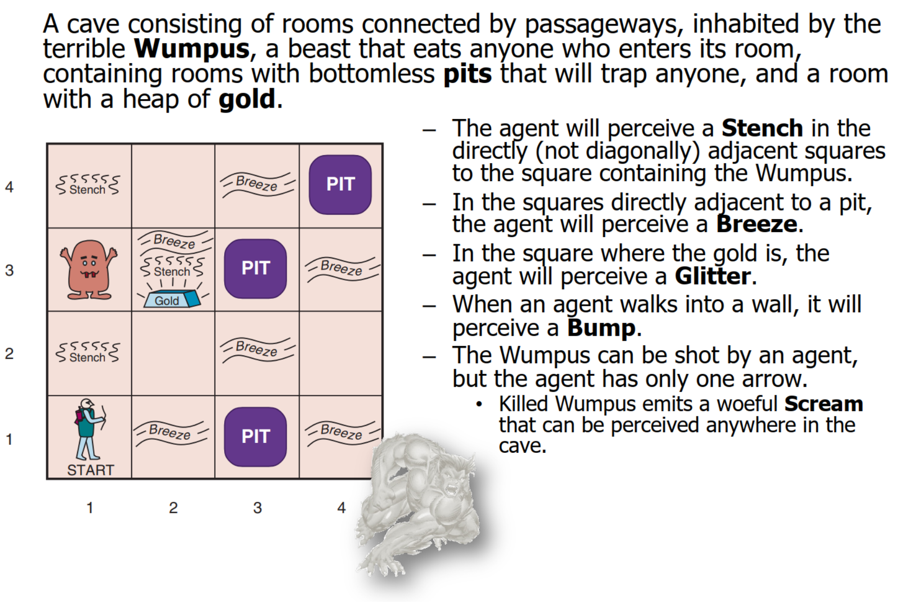
- **Working Principle:**
  - The agent uses **TELL** operation to add new information (from observations or derived inferences) to its knowledge base.
  - The agent uses **ASK** operation to query the knowledge base and ask questions.
  - The agent uses its **inference mechanism** to derive new information, which enables decision making.
  - It combines and recombines information about the world with current observations to uncover hidden aspects of the world and use them for action selection.

- **Significance:** Knowledge-based agents enable sophisticated reasoning, such as deduction and planning. They allow the creation of agents that can operate in complex environments by drawing conclusions from their knowledge.

26.2 The Wumpus World Example

26.2.1 World Description

The Wumpus world is a 4x4 grid of rooms connected by passageways. The key elements are:


A cave consisting of rooms connected by passageways, inhabited by the terrible **Wumpus**, a beast that eats anyone who enters its room, containing rooms with bottomless **pits** that will trap anyone, and a room with a heap of **gold**.

- The agent will perceive a **Stench** in the directly (not diagonally) adjacent squares to the square containing the Wumpus.
- In the squares directly adjacent to a pit, the agent will perceive a **Breeze**.
- In the square where the gold is, the agent will perceive a **Glitter**.
- When an agent walks into a wall, it will perceive a **Bump**.
- The Wumpus can be shot by an agent, but the agent has only one arrow.
  - Killed Wumpus emits a woeful **Scream** that can be perceived anywhere in the cave.

Figure 2: Wumpus World

- **Wumpus**: A dangerous beast that kills anyone entering its room.
- **Pits**: Bottomless pits that trap anyone falling in.
- **Gold**: A valuable item located in a single room.
- **Agent**: The entity exploring the world.

26.3 Sensors and Actuators

The agent can perceive the following:

- **Stench**: In rooms adjacent to the Wumpus.
- **Breeze**: In rooms adjacent to a Pit.
- **Glitter**: In the room with the Gold.
- **Bump**: When the agent walks into a wall.
- **Scream**: Emitted when the Wumpus is killed (can be perceived anywhere).

The agent can perform these actions:

- **MoveForward**: Move one room in the direction facing.
- **TurnLeft**, **TurnRight**: Change the direction facing.
- **Grab**: Pick up the gold.
- **Shoot**: Fire an arrow.
- **Climb**: Climb out of the cave when in the starting position.

26.3.1 Performance Measure

The agent is scored as follows:

- **+1000**: For climbing out of the cave with the gold.
- **-1000**: For falling into a pit or being eaten by the Wumpus.
- **-1**: For each action taken.
- **-10**: For using the arrow.

26.3.2 Logical Representation

Propositional Variables

- $P_{i,j}$: True if there is a pit at location (i, j).
- $B_{i,j}$: True if the agent perceives a breeze at location (i, j).
- $S_{i,j}$: True if the agent perceives a stench at location (i, j).
- $W_{i,j}$: True if the Wumpus is located at (i, j).
- $L_{i,j}$: True if the agent is at location (i, j).

Axioms and Rules

- **No Pit at starting location**: $\neg P_{1,1}$
- **No Wumpus at starting location**: $\neg W_{1,1}$
- **Breeze Rule**: For any location (x,y): $B_{x,y} \Leftrightarrow (P_{x,y+1} \vee P_{x,y-1} \vee P_{x+1,y} \vee P_{x-1,y})$ (where these are only included if these locations are inside the map)
- **Stench Rule**: For any location (x,y): $S_{x,y} \Leftrightarrow (W_{x,y+1} \vee W_{x,y-1} \vee W_{x+1,y} \vee W_{x-1,y})$ (where these are only included if these locations are inside the map)
- **Single Wumpus Rule**: $W_{1,1} \vee W_{1,2} \vee \cdots \vee W_{4,4}$ $\neg W_{1,1} \vee \neg W_{1,2}$ $\neg W_{1,1} \vee \neg W_{1,3}$ $\ldots$ $\neg W_{i,j} \vee \neg W_{k,l}$ for all $i, j \neq k, l$.
- **Agent starting location**: $L_{1,1}$
- **Agent starting direction**: $FacingRight$

26.3.3 Inference Example

In the exploration, these are some of the steps taken:

1. Agent is at [1,1], perceives no stench or breeze. This can be represented as: $\neg S_{1,1}$ and $\neg B_{1,1}$.
2. Moves to [2,1], perceives a breeze : $B_{2,1}$.
3. From $B_{2,1}$, we can deduce that there is a pit at one of these places:[1,1],[2,2] or [3,1]. $B_{2,1} \Rightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$
4. Since we know $\neg P_{1,1}$ then $(P_{2,2} \vee P_{3,1})$
5. Agent at [2,1] there is no breeze in [2,2], then no pit in [2,2] then we infer $P_{3,1}$.

26.4 Propositional Logic

26.4.1 Formulas, CNF, DNF, and Horn Clauses

- **Definition of a Formula:** A formula (or sentence) in propositional logic is built from atomic propositions (variables that can be true or false) and logical connectives.
  - **Atomic Propositions**: Typically denoted by uppercase letters, such as P, Q, R, etc.
  - **Logical Connectives**: $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\Rightarrow$ (implication), $\Leftrightarrow$ (biconditional).
- **Syntax of Propositional Logic:**
  - An atomic proposition is a sentence.
  - If A and B are sentences, then $\neg A$, $A \wedge B$, $A \vee B$, $A \Rightarrow B$ and $A \Leftrightarrow B$ are sentences.
- **Conjunctive Normal Form (CNF):**
  - A formula in CNF is a conjunction of clauses.
  - A clause is a disjunction of literals.
  - A literal is an atomic proposition or its negation.
  - Example: $(A \vee \neg B) \wedge (\neg C \vee D \vee E) \wedge (F)$.
  - **Conversion to CNF:**
    1. Eliminate $\Leftrightarrow$ and $\Rightarrow$: Replace $A \Leftrightarrow B$ with $(A \Rightarrow B) \wedge (B \Rightarrow A)$, and $A \Rightarrow B$ with $\neg A \vee B$.
    2. Move $\neg$ inwards: Apply De Morgan's laws: $\neg(A \wedge B)$ becomes $\neg A \vee \neg B$, and $\neg(A \vee B)$ becomes $\neg A \wedge \neg B$. Also apply double negation: $\neg\neg A$ becomes A.
    3. Distribute $\vee$ over $\wedge$: Apply the distributive laws: $A \vee (B \wedge C)$ becomes $(A \vee B) \wedge (A \vee C)$.
- **Disjunctive Normal Form (DNF):**
  - A formula in DNF is a disjunction of terms.
  - A term is a conjunction of literals.
  - Example: $(A \wedge \neg B) \vee (\neg C \wedge D) \vee (E)$.
  - **Conversion to DNF:** Use the same principles as in CNF conversion but distribute conjunction over disjunction.
- **Horn Clauses:**

- A Horn clause is a disjunction of literals, of which at most one is positive (not negated).
- They can be written in the form $P_1 \wedge P_2 \wedge ... \wedge P_n \Rightarrow Q$ or in clause form $(\neg P_1 \vee \neg P_2 \vee ... \vee \neg P_n \vee Q)$, where P_i and Q are atomic propositions. Note that $n = 0$ is allowed as well.
- Special Cases:
  * **Facts**: A single positive literal (e.g., A).
  * **Rules**: An implication with a conjunction of premises and a single positive conclusion (e.g., $B \wedge C \Rightarrow A$).
  * **Queries**: The target proposition we wish to infer (A) or an empty clause ($\Box$) representing false which means we are seeking proof.

26.4.2 Models, Entailment, Inference, and Satisfiability

- **Model:** A model is an assignment of truth values (true or false) to all propositional variables in a formula, for example $M(A) = true, M(B) = false$ where A and B are propositional variables.
  - If a sentence a is true under an model m, we say that m is a model of a.
  - We use $M(a)$ to denote set of all models of a.
- **Entailment:** A sentence α is entailed by a knowledge base KB, written as $KB \models \alpha$, if and only if, in every model in which the KB is true, the sentence α is also true, i.e. $M(KB) \subseteq M(\alpha)$. I.e. means that α is a logical consequence of KB.
- **Inference:** Inference is the process of deriving new sentences (or conclusions) from existing sentences in the knowledge base.
  - A sound inference algorithm derives only entailed sentences, if $KB \vdash_i \alpha$ then $KB \models \alpha$.
  - A complete inference algorithm can derive all the entailed sentences, if $KB \models \alpha$ then $KB \vdash_i \alpha$.
- **Satisfiability:** A sentence is satisfiable if there exists at least one model in which it is true. Otherwise, the sentence is unsatisfiable.
- **Relations:**
  - α is entailed by KB if the implication $KB \Rightarrow \alpha$ is valid (true in every model).
  - α is entailed by KB if and only if $KB \wedge \neg\alpha$ is unsatisfiable, this is used by many inference algorithms.

26.5 DPLL and WalkSAT Algorithms

26.5.1 DPLL (Davis-Putnam-Logemann-Loveland) Algorithm

- **Purpose:** A sound and complete backtracking algorithm used to check satisfiability of a propositional logic formula in CNF and finds the satisfying model, if one exists.
- **Main Idea:** It performs a depth-first search through the space of possible truth assignments to propositional variables.
- **Heuristics:**
  - **Pure Symbol Heuristic**: If a symbol appears always with the same sign in all clauses, then its assignment is set to a value that satisfies all clauses. If P appears always with P in clauses its value is set to true. If P appears always with $\neg P$ then its value is set to false.

  - **Unit Clause Heuristic**: If a clause is a unit clause (only one literal), then the literal must be set to a value that makes it true.
  - **Early Termination**: Clause is true if any of its literals is true. Formula is not true if any of its clauses is not true.

- **Pseudocode:**

```
Input: clauses, a set of clauses in CNF; symbols, a list of proposition symbols; model,
       a partial model
Output: true if satisfiable, false otherwise
if every clause in clauses is true in model then
    return true
end
if some clause in clauses is false in model then
    return false
end P, value
← FIND-PURE-SYMBOL(symbols, clauses, model) ▷ *[r]Pure symbol heuristic if P is
  not null then
    return DPLL(clauses, symbols - P, model ∪ {P = value})
end P, value
← FIND-UNIT-CLAUSE(clauses, model)     ▷ *[r]Unit clause heuristic if P is not null
  then
    return DPLL(clauses, symbols - P, model ∪ {P = value})
end
P ← FIRST(symbols) rest ← REST(symbols) return DPLL(clauses, rest, model ∪ {P
  = true}) or DPLL(clauses, rest, model ∪ {P = false})                ▷ *[r]Backtracking
```

Algorithm 24: DPLL Algorithm

26.5.2 WalkSAT Algorithm

- **Purpose:** A local search algorithm for satisfiability that is sound but not complete. It does not find a model if no one exists but is faster then DPLL.
- **Main Idea:** It starts with a random assignment of truth values to propositional variables and iteratively tries to improve this assignment by "walking" the solution space by flipping the value of variable.
- **Process:**
  1. Start with a random assignment of truth values to all variables.
  2. If the assignment satisfies all clauses, return it.
  3. If not, randomly select a clause that is not satisfied and randomly or greedily flip a variable from this clause.
  4. Repeat this process for a maximum of max_flips iterations.
- **Pseudocode:**

```
Input: clauses, a set of clauses in propositional logic; p, a probability of "random
       walk" move; max_flips, a number of maximum steps
Output: a satisfying model or failure
model ← random assignment of true/false to the symbols in clauses
for i = 1 to max_flips do
    if model satisfies clauses then
        return model
    end
    clause ← randomly selected clause from clauses that is false in model
    if RANDOM(0, 1) $\leq p$ then
        flip the value in model of a randomly selected symbol from clause
    end
    else
        flip whichever symbol in clause maximizes the number of satisfied clauses
    end
end
return failure
```

Algorithm 25: WalkSAT Algorithm

26.6 Resolution Algorithm

- **Purpose:** A sound and complete refutation-based algorithm that proves entailment or, equivalently, unsatisfiability by applying resolution rules.
- **Resolution Rule:** Given two clauses containing complementary literals, a new clause can be derived by combining both clauses without the complementary literals.
  - If we have two clauses $(A \vee B)$ and $(\neg A \vee C)$ then applying resolution rule gives $(B \vee C)$.
  - A more formal representation with multiple literals:

$$\frac{l_1 \vee \cdots \vee l_{k-1} \vee \mathbf{l_k} \vee l_{k+1} \vee \cdots \vee l_n \quad m_1 \vee \cdots \vee m_{j-1} \vee \neg \mathbf{l_k} \vee m_{j+1} \vee \cdots \vee m_p}{l_1 \vee \cdots \vee l_{k-1} \vee l_{k+1} \vee \cdots \vee l_n \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_p}$$

- **Proving Entailment:**
  - To prove that KB entails α or $KB \models \alpha$, we show that KB $\wedge \neg\alpha$ is unsatisfiable by transforming the expression into CNF.
  - The resolution algorithm takes the clauses of $KB \wedge \neg\alpha$ and repeatedly applies the resolution rule until an empty clause is generated, which indicates unsatisfiability (proving entailment).

- **Pseudocode:**

```
Input: KB, the knowledge base; $\alpha$, the sentence to be checked for entailment
Output: true if KB entails $\alpha$, false otherwise
clauses ← CNF representation of KB $\wedge \neg\alpha$
new ← {}
while true do
    for each pair of clauses $C_i$ and $C_j$ in clauses do
        resolvents ← PL-RESOLVE($C_i$, $C_j$)
        if resolvents contains the empty clause then
            return true                                ▷ *[r]Entailment proved
        end
        new ← new ∪ resolvents
    end
    if new ⊆ clauses then
        return false                                   ▷ *[r]Entailment not proved
    end
    clauses ← clauses ∪ new
end
```

Algorithm 26: Resolution Algorithm

- **Note**: It is often the case that after some applications of resolution rule, two clauses might give a clause that already exists. For example, clauses $(A \vee B)$ and $(\neg A \vee B)$ give a clause (B). However, if the clause (B) already exists, it will not be added. This makes it possible to stop algorithm and conclude that formula is satisfiable.
- We can use a special variant of the resolution algorithm running in **linear time** with respect to the size of KB
- **Forward and Backward Chaining:**
  - **Forward Chaining:**
    - A data-driven approach (starts from known facts).
    - It starts by asserting known facts as true and then using them to infer new facts.
    - It applies modus ponens for Horn clauses, which can be written as $a \wedge b \Rightarrow c$. This is applied by checking premises a and b. If both are known to be true, we can infer that conclusion c is also true.
    - Process stops when no new facts are produced.
    - Good for deriving all consequences from given facts, used in data-driven systems.
  - **Backward Chaining:**
    - A goal-driven approach (starts from the query).
    - Start by considering the query (goal) as a target.
    - Recursively decomposes the query into sub-queries using clauses from KB until the given facts are reached.
    - Good for proving a specific target, used in query systems.

27 Knowledge Representation and First-Order Logic

27.1 Defining a Formula in First-Order Logic

A formula in first-order logic (FOL) is built using several components:

- **Constants**: Represent specific objects in the domain of interest (e.g., 'John', '2', 'The-Crown'). These are typically denoted by lowercase letters, numbers, or capitalized words starting with a letter.

- **Variables**: Represent unspecified objects in the domain (e.g., ‘x‘, ‘y‘, ‘a‘, ‘b‘). They are denoted by lowercase letters, often from the end of the alphabet.
- **Function Symbols**: Represent mappings from objects to objects (e.g., ‘Sqrt‘, ‘LeftLeg‘, ‘Mother‘). They are often represented with lowercase words, the number of arguments is fixed for a particular symbol.
- When applied to terms they represent new objects such as ‘LeftLeg(John)‘.
- **Predicate Symbols**: Represent relationships between objects (e.g., ‘Brother‘, ‘¿‘, ‘OnHead‘). They are often represented with capitalized words, the number of arguments is fixed for a particular symbol.
- When applied to terms, they form atomic formulas such as ‘Brother(Richard,John)‘.
- **Connectives**: Combine simpler formulas into more complex ones:
  - $\neg$ (negation)
  - $\wedge$ (conjunction, AND)
  - $\vee$ (disjunction, OR)
  - $\Rightarrow$ (implication, if-then)
  - $\Leftrightarrow$ (biconditional, if and only if)
- **Equality**: ‘=‘ expresses that two terms refer to the same object (e.g., ‘Father(John) = Henry‘).
- **Quantifiers**: Used to express properties of sets of objects:
  - $\forall$ (universal quantifier) - For all (e.g., $\forall x$ King(x) $\Rightarrow$ Person(x))
  - $\exists$ (existential quantifier) - There exists (e.g., $\exists x$ Crown(x) $\wedge$ OnHead(x, John))

A FOL formula is constructed using the above. Starting with terms (constants, variables and function expressions) then predicate symbols, logical connectives and quantifiers.

Role of Each Component:

- *Constants, variables and function terms* allows to model objects in the domain.
- *Predicate symbols* allows to model relations between objects or properties of objects.
- *Connectives* allows to model complex facts by combining simpler facts.
- *Quantifiers* allows to state facts about a class of objects, not only about a particular one.

Relation between Quantifiers:

- $\neg\forall x P \equiv \exists x \neg P$
- $\neg\exists x P \equiv \forall x \neg P$
- $\forall x \forall y P \equiv \forall y \forall x P$
- $\exists x \exists y P \equiv \exists y \exists x P$
- $\neg\forall x \forall y \equiv \exists x \exists y$
- $\exists x \forall y P \not\equiv \forall y \exists x P$

27.2 Conjunctive Normal Form (CNF)

Definition: A formula in CNF is a conjunction of clauses, where each clause is a disjunction of literals. A literal is an atomic formula (predicate applied to terms) or its negation.

Steps to Obtain CNF:

1. **Eliminate Implications**: Replace $A \Rightarrow B$ with $\neg A \vee B$, $A \Leftrightarrow B$ with $(\neg A \vee B) \wedge (\neg B \vee A)$.
2. **Reduce Scope of Negation**: Use De Morgan's laws: $\neg(A \wedge B) \equiv \neg A \vee \neg B$ and $\neg(A \vee B) \equiv \neg A \wedge \neg B$. Move negations inward, so they only apply to atomic formulas.
3. **Standardization**: Change the names of variables so that quantifiers always refer to different ones. Use different variable names for different quantifiers. For example $\forall x P(x) \vee \exists x Q(x)$ should be transformed into $\forall x P(x) \vee \exists y Q(y)$.
4. **Skolemization**: Remove existential quantifiers:
   - Replace $\exists x P(x)$ by $P(c)$ for a new constant symbol c (Skolem constant).
   - If $\exists$ is inside scope of $\forall$ replace $\exists x P(x, y)$ by $P(f(y), y)$, where f is a new function symbol (Skolem function).
5. **Drop Universal Quantifiers**: Since all variables are now universally quantified, the $\forall$ quantifiers can be dropped (assumed to be the same).
6. **Distribute OR over AND**: Use the distributive law: $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$

27.3 Substitution

Definition: A substitution is a mapping from variables to terms. It is often represented by $\{x_1/t_1, x_2/t_2, \ldots, x_n/t_n\}$, where each x_i is a variable and each t_i is a term. No variable x_i is found in t_j, when i = j.

Applying a Substitution: Given a formula P and a substitution σ, the application of σ to P, written as $P\sigma$, replaces each free variable x_i in P with the corresponding term t_i. For example:

- $P = \text{Knows}(x, y)$
- $\sigma = \{x/John, y/Mary\}$
- $P\sigma = \text{Knows}(John, Mary)$

Unification: Given two formulas (or terms) A and B, unification is the process of finding a substitution σ such that $A\sigma = B\sigma$. Such a σ is called a unifier for A and B. If there exists no such substitution than A and B are not unifiable.

Unification Algorithm: A recursive algorithm to compute the most general unifier (MGU):

```
Input: Terms or formulas A,B
Output: MGU $\sigma$ or FAIL
$\sigma \leftarrow \{\}$;
if $A$ and $B$ are identical then
    return $\sigma$
else
    if $A$ is a variable then
        if $A \in B$ then
            return FAIL
        else
            $\sigma \leftarrow \{A/B\}$ ;
            return $\sigma$;
        end
    else
        if $B$ is a variable then
            if $B \in A$ then
                return FAIL
            else
                $\sigma \leftarrow \{B/A\}$ ;
                return $\sigma$;
            end
        else
            if $A$ and $B$ are predicate symbols and their number of arguments is the same
             then
                For each argument $a_i, b_i$ of A,B call the algorithm. Update MGU $\sigma$.
                 return $\sigma$;
            else
                return FAIL;
            end
        end
    end
end
```

Algorithm 27: Unification Algorithm

27.4 Reduction of FOL to Propositional Logic (Grounding)

Grounding: The process of transforming a FOL formula to a propositional formula, which means eliminating variables, functions and quantifiers. It involves replacing variables with constants from the domain of discourse, the domain of constants should be finite.

Process:

1. **Instantiation**: Generate all possible ground substitutions by assigning all possible constants to variables. E.g. from $\forall x P(x)$ and constants c_1, c_2, generate $P(c_1)$ and $P(c_2)$

2. **Propositionalization**: Treat each ground atomic formula (predicate with constants as arguments) as a propositional variable.

Role of Functions:

- If there are functions in the formula, instantiating variables with constants might not lead to all terms, because instantiating function terms may result in other new terms. The process might be infinite (e.g., $f(a)$, $f(f(a))$, $f(f(f(a)))$ ...).

- Function symbols, when there is infinite domain of them, poses a major problem for a complete reduction to propositional logic. A solution is to limit the depth of function application.

27.5 Resolution Algorithm

Resolution Rule: If a clause contains L and another clause contains $\neg L'$, where L and L' are unifiable, then the clause obtained by resolving these two clauses is a new clause with literals of both clauses except L and $\neg L'$ where the most general unifier σ of L and L' is applied to the obtained literals. Formally stated:

$$\frac{A \vee L, \quad B \vee \neg L'}{(A \vee B)\sigma}$$

Resolution Algorithm:

```
Input: Set of clauses $S$
Output: Empty clause or FAIL
while true do
    Select two clauses $C_1$ and $C_2$ from $S$;
    for all $L_1 \in C_1$, $L_2 \in C_2$ such that $L_1$ is unifiable with $\neg L_2$ do
        Compute most general unifier $\sigma$;
        Compute a new clause $C = (C_1 \setminus L_1)\sigma \vee (C_2 \setminus L_2)\sigma$ ;
        if $C$ is an empty clause then
            return empty clause
        end
        if $C$ is not subsumed by clauses in $S$ then
            $S \leftarrow S \cup C$;
        end
    end
    if All clause pairs were considered then
        return FAIL
    end
end
```

Algorithm 28: Resolution Algorithm

Subsumption: When a clause C contains all literals of other clause D (C is a subset of D) then C subsumes D.

Resolution Strategies: Control the clause selection to make the resolution procedure more efficient:

- **Unit Resolution**: At least one of the parent clauses is a unit clause (a clause with only one literal).
- **Set of Support**: One of the parent clauses is from the set of support (which contains the negation of the goal).
- **Input Resolution**: One of the parent clauses is from the initial set of clauses (the KB).
- **Linear Resolution**: One of the parent clauses is from the initial set of clauses or was generated from a prior resolution step.

Answer Literal: To extract answer from resolution, add $\neg Goal \vee \text{Answer}(x)$ to the clause set, the solution is found by resolving until $\text{Answer}(a)$ is derived (where a is the solution)

27.6 Forward and Backward Chaining

Forward Chaining:

- Starts with known facts and applies inference rules to derive new facts until the goal is reached or no new facts can be derived.
- Useful for bottom-up reasoning.
- Often used in situations where the available data is more readily available than the goal.

- More efficient in cases, where most initial facts participate in the inference.
- Derives all possible facts from the starting facts.

Backward Chaining:

- Starts with the goal and tries to find evidence to support it by looking at facts or simpler sub-goals that can imply this goal.
- Useful for top-down reasoning
- Often used when the goals are well defined and available from the beginning of the problem solving process.
- Useful when just a small part of available facts is necessary for problem solving.
- Only looks for facts supporting the goal

Pattern Matching: The process of finding a match between a pattern (in the premise) with the clauses available in knowledge base. It utilizes the unification algorithm.

- Given pattern P and fact F, compute MGU σ of P and F.
- If there exists a MGU σ, that means P and F match
- Substitute variables in the conclusion by σ and add this new fact to the knowledge base.

Rete Technique:

- Efficient algorithm used in forward chaining to efficiently perform matching of conditions in the premises.
- Constructs a network where each node stores intermediate results that can be used for next matching of different rules.
- When a new fact is derived, it is sent to the top of the network and propagated through.
- If an implication is matched the conclusion is derived and also sent to top of the network.
- Avoids repeatedly matching the same rules by caching intermediate results.

Magic Set:

- Used in backward chaining to prune the number of facts that are used for inference.
- When starting with a goal, a magic set is computed.
- This magic set contains relevant facts, needed to solve the goal.
- When performing backward chaining, look only at the facts in the magic set to solve the goal.

28 Knowledge Engineering

28.1 The Knowledge Engineering Process

The knowledge engineering process is a systematic approach to building intelligent systems that can reason and solve problems using knowledge. It typically involves the following steps:

1. **Identification**:
    - Define the scope of the problem.
    - Identify the goals and objectives of the system.

- Determine the target audience and the intended use of the system.

2. **Conceptualization**:
    - Identify key concepts, relationships, and attributes.
    - Develop a conceptual model or ontology of the domain.
    - Use various knowledge representation techniques to model the domain.

3. **Formalization**:
    - Convert the conceptual model into a formal representation.
    - Choose an appropriate knowledge representation language (e.g., first-order logic, frames, semantic networks).
    - Define axioms and rules for reasoning with the knowledge.

4. **Implementation**:
    - Implement the knowledge base using a suitable software tool or language.
    - Develop reasoning algorithms and inference mechanisms.
    - Handle specific computational or logical limitations.

5. **Testing**:
    - Validate the system with test cases and scenarios.
    - Verify the system's performance against the goals.
    - Debug and refine the knowledge base and reasoning mechanisms.

6. **Maintenance**:
    - Monitor the system's performance over time.
    - Update the knowledge base as new information becomes available.
    - Adapt the system to changing needs and requirements.

28.2 Efficient Representation of Objects (Taxonomy)

A taxonomy is a hierarchical classification of objects. It allows for efficient organization and retrieval of information. Key principles for efficient representation include:

1. **Inheritance**:
    - More specific concepts (subclasses) inherit properties from more general concepts (superclasses).
    - Reduces redundancy and promotes a more compact representation.

2. **Generalization**:
    - Forming more general concepts from specific instances by removing unique details.
    - Helpful in grouping similar items to save on information storage.

3. **Specialization**:
    - Creates more specific categories by adding attributes or limitations to existing classes.
    - Useful for when more specific distinctions are needed.

4. **Is-a and Instance-of relationships**:
    - Used to define the hierarchy of concepts and the relations between classes.
    - "Is-a" represents a subclass relationship and "instance-of" indicates that a specific object belongs to a particular class.

Concept	Description
Superclass	A general category of objects
Subclass	A more specific category of objects that inherits from a superclass
Instance	A specific object that belongs to a class
Is-a	Relation defining a subclass relationship
Instance-of	Relation indicating a specific object's class membership

Table 3: Key Concepts in Taxonomy

28.3 Situation Calculus

Situation calculus is a formalism used for representing actions, time, and changes in the world. It focuses on describing *situations* that are changed by actions.

28.3.1 Representing Situations

- A situation is a snapshot of the world at a specific time, denoted by variables like s.
- The initial state of the world is usually denoted by S_0.
- Actions take us from one situation to another. The result of action a performed in situation s is a new situation: $do(a, s)$.

28.3.2 Fluent and Rigid Predicates

- **Fluent Predicates**:
  - Represent properties that can change over time or with actions.
  - Example: `On(BlockA, Table, s)`, meaning Block A is on the table in situation s.
- **Rigid Predicates**:
  - Represent properties that do not change over time or with actions.
  - Example: `IsBlock(BlockA)`, meaning Block A is always a block, regardless of situation.

28.3.3 Axioms in Situation Calculus

1. **Possibility Axiom**:
   - Determines when an action can be performed in a given situation.
   - Example: `Poss(Move(BlockA, Table),s)` $\implies$ $\neg$ `On(BlockA,Table,s)` (You can move a block to a table if it's not already on the table).
   - General form: $Poss(a, s) \implies \text{condition}(s)$
2. **Effect Axiom**:
   - Specifies the changes in the world when an action is performed.
   - Example: `Poss(Move(BlockA,Table),s)` $\implies$ `On(BlockA,Table, do(Move(BlockA,Table),s))`.
   - General form: $Poss(a, s) \implies P(do(a, s))$ (for fluent P).
3. **Frame Axiom**:
   - States what doesn't change after an action.
   - Example: `On(BlockB, Floor,s)` $\wedge$ $\neg$ `(a = Move(BlockB,x))` $\implies$ `On(BlockB,Floor, do(a,s))`, if BlockB is on the floor, and we don't move it, then it is still on the floor.

- General form: $P(s) \wedge \neg\text{cause}(a, P) \implies P(do(a, s))$

4. **Successor-State Axiom**:
   - Combines the effects and frame axioms into a single statement.
   - Example:

$$On(BlockA, Table, do(a, s)) \iff$$
$$(a = Move(BlockA, Table) \vee (On(BlockA, Table, s) \wedge \neg(\exists x.a = Move(Block$$

   - General form: $P(do(a, s)) \iff \text{Effect} \vee (P(s) \wedge \neg\text{cause}(a, P))$

28.4 Frame and Ramification Problems

28.4.1 Frame Problem

- **Definition**: The difficulty of specifying all the things that don't change after an action.
- **Challenge**: Writing explicit frame axioms for every fluent predicate and action is tedious and leads to a large, cumbersome knowledge base.
- **Solutions**:
  - **Successor-State Axioms**: Combine frame and effect axioms to concisely specify changes.
  - **Explanation Closure**: Assume that if a fluent changes, then there exists an action that caused it.

28.4.2 Ramification Problem

- **Definition**: The difficulty in determining indirect effects of actions.
- **Challenge**: Actions can have secondary effects that are not explicitly stated in the effect axioms.
- **Example**: If you pick up a block, all the blocks on it move as well. This needs to be encoded.
- **Solutions**:
  - **Causal Rules**: Add rules that specify how changes in one fluent can cause changes in other fluents.
  - **Fluent Dependencies**: Use dependencies to describe interconnected changes to the system and fluents.

29 Introduction to Automated Planning

29.1 What is Planning?

Planning is a crucial aspect of intelligent behavior. It involves the process of finding a sequence of actions that, when executed, will achieve a desired goal. In essence, it is about reasoning about the future and deciding what to do to reach a specific state.

29.2 Planning vs. Projection

- **Planning:** The task of finding a sequence of actions (a plan) to achieve a goal, given an initial state. It involves reasoning about how to change the world to reach a desired state.
- **Projection:** The task of predicting the outcome of a given sequence of actions. It involves simulating the execution of a plan to determine its final state.

30 Classical Representation of Planning Problems

30.1 States

- A **state** represents a snapshot of the world at a particular point in time.
- In classical planning, a state is a set of **instantiated atoms** (ground literals).
- Atoms are predicates with specific constant values.
- The number of possible states is finite.
- **Closed World Assumption:** If an atom is not in the state, it is assumed to be false.
- **Fluents:** Atoms that can change their truth value across states (e.g., `at(robot,loc1)`).
- **Rigid Atoms:** Atoms that have the same truth value in all states (e.g., `adjacent(loc1,loc2)`).

30.2 Operators vs. Actions

- **Operator:** A schema defining a class of actions. It consists of:
  - **name(o):** The operator's name with variables (e.g., `move(r,l,m)`).
  - **precond(o):** Literals that must hold in the state for the operator to be applicable.
  - **effects(o):** Literals that become true after applying the operator (only fluents).
- **Action:** A fully instantiated operator, where all variables have been replaced with constant values (e.g., `move(robot1,loc1,loc2)`).

30.3 Transition Function

- The **transition function** $\gamma(s, a)$ defines how a state changes when an action is applied.
- If action a is applicable to state s:
  - $\gamma(s, a) = (s - effects^{-}(a)) \cup effects^{+}(a)$
  - $effects^{+}(a)$ are the positive effects of the action a.
  - $effects^{-}(a)$ are the negative effects of the action a.
- If action a is not applicable to state s, $\gamma(s, a)$ is undefined.

30.4 Goal Function

- The **goal function** g is a set of literals that define the desired state.
- A state s satisfies the goal g if:
  - $g^{+} \subseteq s$
  - $g^{-} \cap s = \emptyset$
- g^{+} are the positive literals in the goal.
- g^{-} are the negative literals in the goal.

30.5 Planning Domain

A planning domain Σ is defined as a triple (S, A, γ), where:

- S is the set of all possible states.
- A is the set of all possible actions.
- γ is the transition function.

30.6 Planning Problem

A planning problem P is defined as a triple (Σ, s_0, g), where:

- Σ is the planning domain.
- s_0 is the initial state.
- g is the goal condition.

A solution to the planning problem is a sequence of actions (a plan) $\pi = \langle a_1, a_2, \ldots, a_k \rangle$ such that $\gamma^*(s_0, \pi)$ satisfies the goal condition g.

31 State-Space Planning

31.1 Overview

State-space planning explores the space of possible states to find a path from the initial state to a goal state. It treats the planning problem as a search problem in the state space.

31.2 Forward Planning (Progression)

- Starts with the initial state s_0.
- Applies applicable actions to generate successor states.
- Continues until a goal state is reached.

```
Input  : Set of operators $O$, Initial state $s_0$, Goal $g$
Output: A plan $\pi$ or failure
$s \leftarrow s_0$;
$\pi \leftarrow$ empty plan;
while true do
    if $s$ satisfies $g$ then
        return $\pi$;
    end
    $E \leftarrow \{a \mid a$ is a ground instance of an operator in $O$ and $precond(a)$ is true in $s\}$;
    if $E = \emptyset$ then
        return failure;
    end
    nondeterministically choose an action $a \in E$;
    $s \leftarrow \gamma(s, a)$;
    $\pi \leftarrow \pi.a$;
end
```

Algorithm 29: Forward-search(O, s_0, g)

31.3 Backward Planning (Regression)

- Starts with the goal condition g.
- Identifies relevant actions that can achieve parts of the goal.
- Regresses the goal by applying the action's preconditions.
- Continues until a sub-goal that is satisfied by the initial state is reached.

```
Input  : Set of operators $O$, Initial state $s_0$, Goal $g$
Output: A plan $\pi$ or failure
$\pi \leftarrow$ empty plan;
while true do
    if $s_0$ satisfies $g$ then
        return $\pi$;
    end
    $A \leftarrow \{a \mid a$ is a ground instance of an operator in $O$ and $\gamma^{-1}(g, a)$ is defined$\}$;
    if $A = \emptyset$ then
        return failure;
    end
    nondeterministically choose an action $a \in A$;
    $\pi \leftarrow a.\pi$;
    $g \leftarrow \gamma^{-1}(g, a)$;
end
```

Algorithm 30: Backward-search(O, s_0, g)

31.4 Lifting

In backward planning, instead of using ground actions, we can use partially instantiated actions and use unification to find relevant actions.

32 Plan-Space Planning

32.1 Overview

Plan-space planning searches the space of plans instead of the space of states. It starts with an initial partial plan and refines it until a complete plan is found.

32.2 Partial Plan

A partial plan Π is defined as a tuple $(A, <, B, L)$, where:

- A is a set of partially instantiated planning operators (actions).
- $<$ is a partial order on A ($a_i < a_j$ means a_i must be executed before a_j).
- B is a set of constraints on variables in the actions (e.g., $x = y$, $x \neq y$, $x \in D_i$).
- L is a set of causal relations ($a_i \rightarrow^p a_j$ means a_i achieves precondition p of a_j).

32.3 Initial Plan

The initial plan contains two special actions:

- a_0: Represents the initial state (effects are the initial state, no preconditions).
- a_∞: Represents the goal (preconditions are the goal, no effects).

32.4 Solution Plan

A partial plan is a solution plan if:

- There are no flaws (no open goals and no threats).
- The partial ordering and constraints are globally consistent.

32.5 Open Goal

An **open goal** is a precondition p of some action b in the partial plan for which there is no action that achieves it (no causal relation $a \rightarrow^p b$). Open goals can be resolved by:

- Finding an action a that can achieve p (p is among the effects of a).
- Adding a causal relation $a \rightarrow^p b$.
- Binding the variables from p.

32.6 Threat

A **threat** is an action b that can influence an existing causal relation $a_i \rightarrow^p a_j$. If an action b has effects that can negate p and b can be between a_i and a_j, then b is a threat. Threats can be resolved by:

- Ordering b before a_i.
- Ordering b after a_j.
- Binding variables in b so that it does not negate p.

```
**Input**  : Partial plan $\pi$
**Output:** A solution plan or failure
flaws ← OpenGoals($\pi$) ∪ Threats($\pi$);
if $flaws = \emptyset$ then
    return $\pi$;
end
select any flaw $\phi \in$ flaws;
resolvers ← Resolve($\phi$, $\pi$);
if $resolvers = \emptyset$ then
    return failure;
end
nondeterministically choose a resolver $\rho \in$ resolvers;
$\pi'$ ← Refine($\rho$, $\pi$);
return PSP($\pi'$);
```

Algorithm 31: PSP(π)