Imperial College
London

NOTES

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Introduction To Artificial Intelligence

Author:
Your Name (CID: your college-id number)

Date: April 23, 2018

1 Introduction to AI

Artificial Intelligence is the study and development of intelligence software agents, machines that act intelligently, that manifest some of the capabilities that the human mind has.

1.1 Knowledge Representation

There are different forms of representation of knowledge. These are referred to as **knowledge schemas**.

In domains of certainty, a symbolic, **logic-based form** of representation of knowledge is appropriate. In domains of uncertainty, **probabilistic form** of representation of knowledge might be more suitable.

Once we fix a representation schema, then the formalisation of a piece of knowledge in that schema is called **representation** of the knowledge.

The representation of the entire knowledge that is stored in an agent is instead referred to as **knowledge base** of the agent.

A representation of the knowledge should have the following properties:

1. Rich enough to express the knowledge needed to solve the problem.

2. Close to the problem

3. Amenable to efficient computation, which means it should be able to trade computation time with accuracy.

4. Able to be acquired from people, data and past experiences.

The representation of the problem constitutes essentially a **model** of the world. The model specifies what is true and false in the world.

All models are **abstractions**. A lower-level abstraction includes more detail than a higher-level abstraction.

High level abstraction is easier for a human to understand the system's solution. But it is harder for a machines to fully comprehend due to the high level of ambiguity.

Low level abstraction can be more accurate and more predictive, the it is also more difficult to reason with. This is because low-level abstractions are too detailed and environment changes at micro-level more than at abstract level.

1.2 Defining Solution

To solve a problem, we must first define the task and what constitutes a solution.

Optimal Solution: This are the ones that are the best solutions according to some criteria for measuring the quality of solution

Satisfying Solution: This are the ones that are good enough according to some notion of what is an adequate solution

Approximately optimal solution: This are the ones that are close enough to the optimal solution

Probable solution: This are the ones that may not actually be a solution to the problem, but it is likely to be a solution.

1.3 Reasoning

Reasoning is the process made by a computational agent when searching for possible ways to determine to to complete its task.

Offline reasoning assumes that the background knowledge needed to solve the task is precomputed

Online reasoning allow agents to sense the environment to collect online data during the execution. It then uses the collected data and the given background knowledge to decide what actions to take

A reasoning process can be of different types: **Deductive**, **Abductive** and **Inductive**.

In deductive inferences, what is inferred is necessarily true if the premises that it is inferred from are true. Abductive and inductive inference are not necessarily true.

Deduction uses the principle that true premises yield to true conclusions. **Resolution** is a mechanism for computing this inference step.

Induction is the process of generalisation. It can be seen as the reverse of deduction because it amplifies the generality of observations into general learned rules.

Abduction is the process of explanation. It starts from the general rule and the observations, and then find possible situations in which the given rules would generate the given observation.

1.4 Computational Agents

There are some key aspects that we will need to take into account when we develop a computational agent.

1. **Model of the Environment**: These can be expressed in terms of states, features and relations between individuals

2. **Uncertainty**: Uncertainty on the **state** can be **full observable state** or **partially observable state**. In Full observable state, the agent know exactly which state it is in. In Partially observable state, there are multiple possible state that can lead to the

same agent's perception.

Uncertainty on the **actions** can be **deterministic** or **stochastic**. Deterministic means the agent knows the exact state after it performs the action. Stochastic means the agent may reach more than one possible state after the action

3. **Preference**: These occur when there are trade off between the desirability of possible outcomes.

2 Searching

This chapter describes one kind of goal-based agent called a **problem-solving agent**. Problem-solving agents use **atomic** representations, meaning states of the world are considered as whole, with no internal structure visible to the problem-solving algorithms.

In this chapter, we limit ourselves to the simplest kind of task environment, for which the solution to a problem is always a fixed sequence of actions.

In particular, the environment must be:

1) **Observable**, so the agent always knows the current state.

2) **Discrete**, so at any given state there are only finitely many actions to choose from.

3) **Known**, so the agent knows which states are reached by each action.

4) **Deterministic**, so each action has exactly one outcome.

2.1 Well Defined Problems

A **problem** can be defined formally by five components:

1) The **initial state** that the agent starts in.

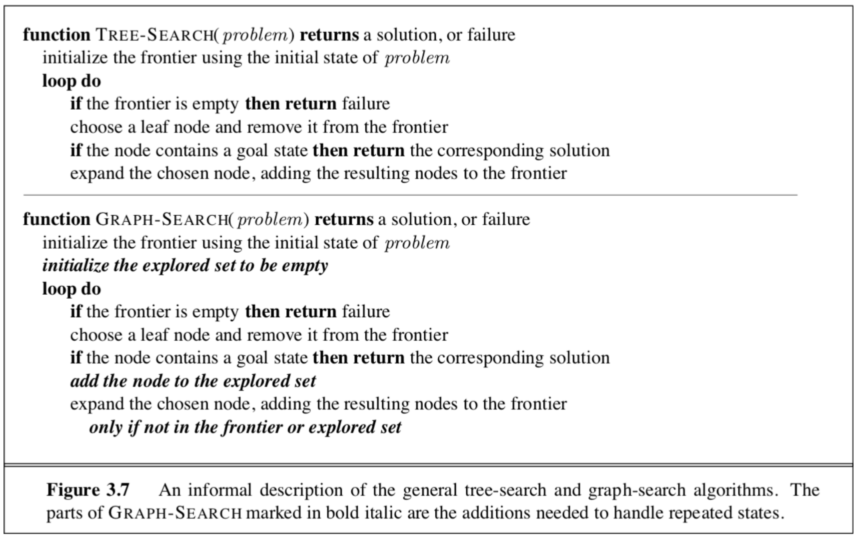
2) A description of the possible **actions** available to the agent. Given a particular state S, $action(S)$ returns a set of actions that can be executed in S. We say that each of these actions is **applicable** in S.

3) The **transition model**, which is a description of what each action does. This is specified by a function $result(S, A)$ that returns the state that that results from doing action A in S. We use the term **successor** to refer to any state reachable from a given state by a single action.

4) The **goal test** determines whether a given state is a goal state.

5) A **path cost** function that assigns a numerical cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure (cost, time, distance). The **step cost** of taking action A in state S to reach state S' is denoted by $c(S, A, S')$.

2.2 Searching for Solution



```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
     if the frontier is empty then return failure
     choose a leaf node and remove it from the frontier
     if the node contains a goal state then return the corresponding solution
     expand the chosen node, adding the resulting nodes to the frontier
```

```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
     if the frontier is empty then return failure
     choose a leaf node and remove it from the frontier
     if the node contains a goal state then return the corresponding solution
     add the node to the explored set
     expand the chosen node, adding the resulting nodes to the frontier
        only if not in the frontier or explored set
```

Figure 3.7 An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

2.2.1 Tree Search

The possible action sequences starting at the initial state from a **search tree** with the initial state at the **root**. The **branches** are actions and the **nodes** correspond to states in the state space of the problem.

The first step is to test whether the root is a goal state. If not, we need to **expand** the current state by applying each legal action to the current state to **generate** a new set of states. Adding new **child nodes** to the **parent node**. The set of all **leaf nodes** available for expansion at any given point is called the **frontier**.

The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand. Search algorithms all share this basic structure; they vary primarily according to how they choose which state to expand next - the so-called **search strategy**.

2.2.2 Graph Search

Redundant paths exist whenever there is more than one way to get from one state to another. An example of redundant path is **loopy path** when the search expands on a **repeated state** in the search tree. The way to avoid exploring redundant paths is to remember where one has been. To do this, we augment the tree search algorithm with a data structure called **explored set** which remembers every expanded node. The new algorithm is called **graph-search**.

The search tree constructed by graph-search algorithm contains at most one copy of each state. This algorithm has a nice property: the frontier separates the state-space graph into the explored region and the unexplored region, so that every path from the initial state to an unexplored state has to pass through a state in the frontier.

2.2.3 Measuring problem-solving performance

We can evaluate an algorithm's performance in four ways:

1) **Completeness**: The algorithm is guaranteed to find a solution when there is one

2) **Optimality**: The strategy finds the optimal solution

3) **Time complexity**: The time is takes to find a solution

4) **Space complexity**: The memory needed to perform the search

In AI, the graph is often represented implicitly by the initial state, actions and transition model and is frequently infinite. For these reasons, complexity is expressed in terms of three quantities:

1) b, the **branching factor** or the maximum number of successors of any node

2) d, the **depth** of the shallowest goal node

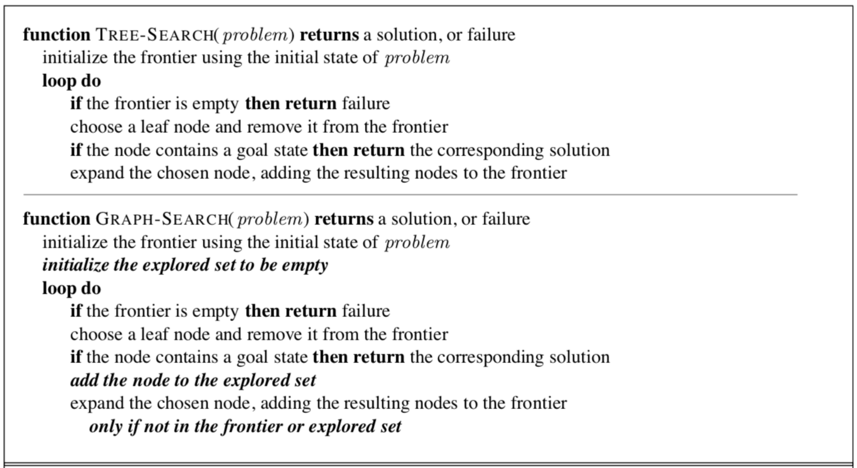
3) m, the **maximum length** of any path in the state space.

Time is often measured in terms of the number of nodes generated during the search. Space is measured in terms of the maximum number of nodes stored in memory.

2.3 Uninformed Search Strategies

Uninformed search are also called **blind search**. The term means that the strategies have no additional information about states beyond that provided in the problem definition. All they can do is generate successors and distinguish a goal state from a non-goal state

2.3.1 Breadth-first Search



```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
      if the frontier is empty then return failure
      choose a leaf node and remove it from the frontier
      if the node contains a goal state then return the corresponding solution
      expand the chosen node, adding the resulting nodes to the frontier

function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
      if the frontier is empty then return failure
      choose a leaf node and remove it from the frontier
      if the node contains a goal state then return the corresponding solution
      add the node to the explored set
      expand the chosen node, adding the resulting nodes to the frontier
         only if not in the frontier or explored set
```

Figure 3.7 An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

Breadth first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, the their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

Breadth first search is **complete**. If the shallowest goal node is at some finite depth d, breadth first search will eventually find it after generating all shallower nodes (provided b is finite).

As soon as a goal node is generated, we know it is the shallowest goal node because all shallower nodes must have been generated already and failed the goal test. The shallowest goal node does not mean it is optimal. Breadth first search is only **optimal** if the path cost is a nondecreasing function of the depth of the node.

The total number of nodes generated in breadth first search, or its **time complexity** is $b + b^2 + b^3 + ... + b^d = O(b^d)$. The root generates b nodes at the first level, then each node generates b successors, making a total of b^2 at the second level and so on.

For breath first search, every node generated remains in memory. There will be $O(b^{d-1})$ nodes in the explored set and $O(b^d)$ nodes in the frontier. So the **space complexity** is $O(b^d)$.

2.3.2 Uniform-cost Search

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
      if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          if child.STATE is not in explored or frontier then
              frontier ← INSERT(child, frontier)
          else if child.STATE is in frontier with higher PATH-COST then
              replace that frontier node with child
```

Figure 3.13 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure **??**, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

Uniform-cost search expands the node n with the lowest path cost $g(n)$. This is done by storing the frontier as a priority queue ordered by g.

In uniform cost search, the goal test is applied to a node when it is selected for expansion, rather than it is first generated. The reason is that the first goal node that is generated may be on a suboptimal path hence not the optimal solution.

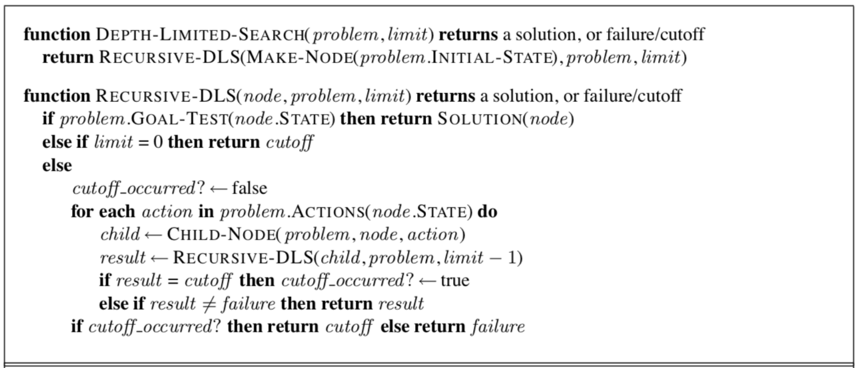
Furthermore, a test is added in case a better path is found to a node currently on the frontier.

Uniform cost search does not care about the number of steps a path has, but only about the total cost. Therefore it will get stuck in an infinite loop if there is a path with infinite sequence of zero-cost actions. This algorithm is only **complete** provided the cost of every step exceeds some small positive constant ϵ.

The uniform cost search is **optimal**. We observe that when ever the algorithm selects a node n for expansion, the optimal path to that node has been found. Also because step costs are nonnegative, paths never get shorter as nodes are added. Hence the first node selected for expansion must be optimal

Uniform cost search is guided by path costs rather than depths, so its complexity is not easily characterised in terms of b and d. Instead, let $C*$ be the cost of the optimal solution, and assume every action costs at least ϵ. The the worst-case **time and space complexity** is $O(b^{1+\frac{C*}{\epsilon}})$.

2.3.3 Depth-first Search



```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Figure 3.16 A recursive implementation of depth-limited tree search.

Depth first search always expands the deepest node in the current frontier of the search tree.

In **finite state space**, depth first search is **complete** in the graph-search version, which avoids repeated sates and redundant paths, because it will eventually expand every node. On the other hand, depth first search is **not complete** in the tree-search version because it can loop forever. In **infinite state spaces**, both versions are **incomplete** if an infinite non-goal path is encountered.

Depth first search is **not optimal** because it may return a goal state that is deeper in the search tree.

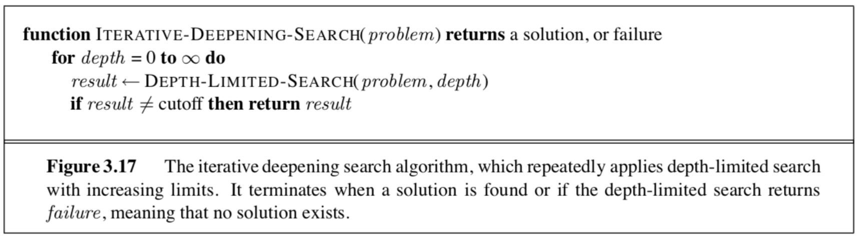
The **time complexity** of a depth first graph search is bounded by the size of the state space. A depth first tree search may generate all of the $O(b^m$ nodes in the search tree.

In terms of **space complexity**, depth first graph search has no advantage. However, a depth first tree search needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path, this means it only requires storage of $O(bm)$ nodes.

The embarrassing failure of depth first search in infinite state space can be alleviated by supplying depth first search with a predetermined depth limit l. This approach is called **depth-limited search**.

The search is **not complete** if we choose $l < d$ meaning the shallowest goal is beyond the depth limit. The search is **non optimal** if we choose $l > d$. Its time complexity is $O(b^l)$ and its space complexity is $O(bl)$.

2.3.4 Iterative deepening Search



```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
```

Figure 3.17 The iterative deepening search algorithm, which repeatedly applies depth-limited search with increasing limits. It terminates when a solution is found or if the depth-limited search returns *failure*, meaning that no solution exists.

Iterative deepening search is a general strategy, often used in combination with depth first search, that finds the best depth limit. It does this by gradually increasing the limit - first 0, then 1, then 2 and so on - until a goal is found. This will occur when the depth limit reaches d, the depth of the shallowest goal node.

Iterative deepening search combines the benefits of depth first and breadth first search. Like depth first search, its **memory complexity** is $O(bd)$. Like breadth first search, it is **complete** when b is finite and **optimal** when the path costs is a nondecreasing function of the depth of the node.

In an iterative deepening search, the nodes on the bottom level are generated once, those on the next-to-bottom level are generated twice and so on. So the total number of nodes generated in worst case is $db + (d-1)b^2 + (d-2)b^3 + ... + b^d$ that gives **time complexity** $O(b^d)$.

2.3.5 Bidirectional Search

The idea behind bidirectional search is to run two simultaneous searches - one forward from the initial state and the other backward from the goal - hoping that the two searches meet in the middle.

Bidirectional search is implemented by replacing the goal test with a check to see whether the frontiers of the two searches intersect; if they do, a solution has been found.

The **time complexity** of bidirectional search using breadth first searches in both direction is $O(b^{\frac{d}{2}})$. The **space complexity** is $O(b^{\frac{d}{2}})$.

2.4 Informed Search Strategies

An informed search strategy is one that uses problem-specific knowledge beyond the definition of the problem itself.

A general approach we consider is called **best-first search**. Best-first search is an instance of the general tree-search or graph-search algorithm in which a node selected for expansion based on a **evaluation function**, $f(n)$. The evaluation function is constructed as a cost estimate so the nodes with the lowest evaluation is expanded first.

The choice of $f(n)$ determines the search strategy. Most best-first algorithm include as a component of f a **heuristic function**, $h(n)$. $h(n)$ is the estimated cost of the cheapest path from the state at node n to a goal state.

2.4.1 Greedy Best-first Search

Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus is evaluates nodes by using just the heuristic function, $f(n) = h(n)$.

Greedy best-first search is **not complete** because it may go into an infinite loop when it reaches a dead end. It does not know how to get to the goal by taking another longer path.

Greedy best-first search is **not optimal**, at each step it tries to get as close to the goal as possible even if it has a higher path cost.

The worst case **time and space complexity** for the tree version is $O(b^m)$

2.4.2 A* Search

The most widely known form of best-first search is called A* search. It evaluates nodes by combining the cost to reach the node $g(n)$, and $h(n)$ the cost to get from the node to the goal. This means $f(n) = g(n) + h(n)$.

For A* search to be both **complete** and **optimal**, the heuristic function must satisfy two conditions:

1) $h(n)$ has to be an **admissible heuristic**, this means it never overestimates the cost to reach the goal. Admissible heuristics are by nature optimistic because they think the cost of solving the problem is less than it actually is.

2) A slightly stronger condition called **consistency**. 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'. $h(n) \leq c(n, a, n') + h(n')$.

2.4.3 Heuristic Functions

A heuristic function can be generated from a **relaxed problem**, which is a problem with fewer restrictions on the actions.

Because the relaxed problem adds edges to the state space, any optimal solution in the original problem is also a solution in the relaxed problem; but the relaxed problem may have better solutions if the added edges provide short cuts. Hence the cost of an optimal solution to a relaxed problem is an **admissible** heuristic for the original problem. Furthermore, because the derived heuristic is an exact cost for the relaxed problem, it must be **consistent**.

An example of heuristic function to a route-finding problem is the straight line distance between two points. Straight line distance is admissible because the shortest path between any two points is a straight line, so the line cannot be overestimated.

3 Planning

The problem-solving agent that uses searching can find sequences of actions that result in a goal state. But it deals with atomic representations of states and thus needs good domain-specific heuristics to perform well.

In response to this, planning researchers have settled on a **factored representation**, one in which a state of the world is represented by a collection of variables.

3.1 Well Defined Problems

Each **state** is represented as a conjunction of fluent that are ground, functionless atoms. $In(x)$ cannot be part of a state because it is not ground. $\neg Dirt(A)$ cannot be part of a state because it is not an atom. $In(Adjacent(A))$ cannot be part of a state because it contains a function. Fluents not ?mentioned? in a state representation are (implicitly) false.

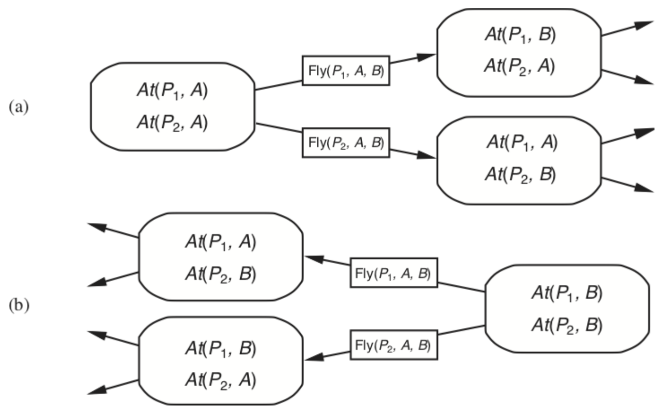
Actions are described by a set of action schemas that implicitly define the $ACTION(s)$ and $RESULT(s,a)$ functions needed to do a problem-solving search. A set of ground actions can be represented by a single **action schema**. The schema consists of the action name, a list of all the variables used in the schema, a **precondition** and an **effect**. We say an action a is **application** in state s if the preconditions are satisfied by s.

The **result** of executing action a in state s is defined as a state s' which is represented by the set of fluents formed by starting with s, removing the fluents that appear as negative literals in the action's effect, **delete list**, and adding the fluents that are positive literals in the action's effect, **add list**. $RESULT(s,a) = (s - DEL(a)) \cup ADD(a)$. Fluents persist (do not change) unless they are explicitly changed by an action.

The **goal** is just like a precondition: a conjunction of literals (positive or negative) that may contain variables, such as $At(p,SFO) \wedge Plane(p)$. Any variables are treated as existentially quantified, so this goal is to have any plane at SFO. A goal cannot be universally quantified, such as $\forall x \neg Dirt(x)$.

3.2 Searching for Solution

The description of a planning problem defines a search problem: we can search from the initial state through the space of states, looking for a goal. One of the nice advantages of the declarative representation of action schemas is that we can also search backward from the goal, looking for the initial state.



There are two approaches of searching for a plan: (a) Forward (progressive search). (b) Backward (regression) search.

3.2.1 Forward (progression) state-space search

Forward search starts in the initial state and using the problem's actions to search forward for a member of the set of goal states.

Forward search is prone to exploring irrelevant actions. Planning problems often have large state space. Therefore forward search is hopeless without an accurate heuristic.

3.2.2 Backward (regression) relevant-state search

In regression search we start at the goal and apply the actins backward until we find a sequence of steps that reaches the initial state. It is called **relevant-states** search because we only consider actions that are relevant to the goal.

Backward search works only when we know how to regress form a state description to the predecessor state description. Given a ground goal description g and a ground action a, the regression from g over a gives us a state description g' given by $g' = (g - ADD(a)) \cup Precond(a)$. Note that $DEL(a)$ does not appear because while we know the fluents in $DEL(a)$ are no longer true after the action, we don't know whether or not they were true before.

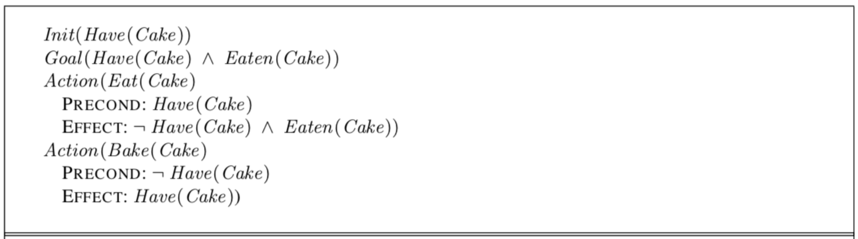
To get the full advantage of backward search, we need to deal with partially uninstantiated actions and states, not just ground ones. We can take advantage of the power of first-order representations. For example, to summarise the possibility of using any of the planes, $g' = In(C_2, p) \wedge At(p, SFO) \wedge Cargo(C_2) \wedge Plane(p) \wedge Airport(SFO)$.

The final issue is deciding which actions are candidates to regress over. In backward search we want actions that are relevant. For an action to be relevant to a goal it obviously must contribute to the goal: at least one on the action's effects must unify with an element of the goal.

3.3 Planning Graphs

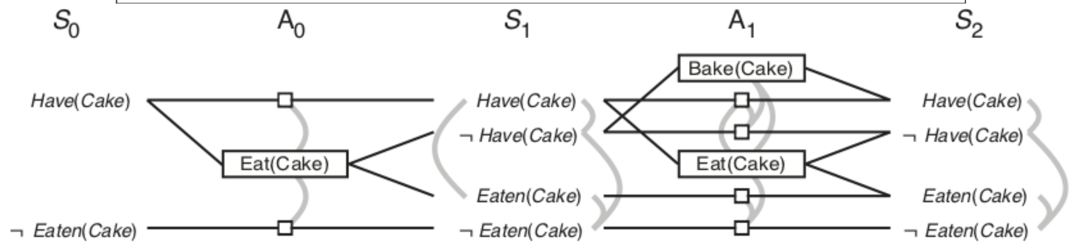
Neither forward nor backward search if efficient without a good heuristic function. A heuristic function estimates the distance from state s to the goal, and if we can derive an admissible heuristic for this distance then we can use A* search to find optimal solutions.

An admissible heuristic can be derived by defining a relaxed problem that is easier to solve, however these heuristics can suffer from inaccuracies. A special data structure called **planning graph** can be used to give better heuristic.



```
Init(Have(Cake))
Goal(Have(Cake) ∧ Eaten(Cake))
Action(Eat(Cake)
  PRECOND: Have(Cake)
  EFFECT: ¬ Have(Cake) ∧ Eaten(Cake))
Action(Bake(Cake)
  PRECOND: ¬ Have(Cake)
  EFFECT: Have(Cake))
```

Figure 10.7 The "have cake and eat cake too" problem.



A planning graph is a directed graph organised into **levels**: first a level S_0 for the initial state, consisting of nodes representing each fluent that holds in S_0; then a level A_0 consisting of nodes for each ground action that might be applicable in S_0; then alternating levels S_i followed by A_i; until we reach a termination condition.

Each action at level A_i is connected to its preconditions at S_i and its effects at S_{i+1}. So a literal appears because an action caused it, but we also want to say that a literal can persist if no action negates it. The is represented by a **persistence action**, drawn as small square boxes.

Level A_0 contains all the actions that could occur in state S_0. The grey lines indicate **mutual exclusion** (or **mutex**) links. Level S_i contains all the literals that could result from picking any subset of the actions in A_0. We continue in this way, alternating between state level and action level until we reach a point where two consecutive levels are identical. At this point, we say that the graph has **leveled off**.

3.3.1 Defining mutex links

A mutex relation hold between two **actions** at a given level if any of the conditions hold:

1) **Inconsistent effects**: one action negates the an effect of another. For example, the action $Eat(Cake)$ negates the effect of the persistence action of $Have(Cake)$.

2) **Interference**: one of the effects of the action is the negation of a precondition of the other. For example, the effect of $Eat(Cake)$ negates the precondition of the persistence action of $Have(Cake)$.

3) **Competing needs**: one of the preconditions of one action is mutually exclusive with a precondition of the other. For example, the precondition of $Bake(Cake)$ and precondition of $Eat(Cake)$ is mutually exclusive.

A mutex relation hold between two **states** at a given level if any of the conditions hold:

1) The two literals at the same level is the negation of the other. For example $Have(Cake)$ and $\neg Have(Cake)$.

2) For each possible pair of actions that could achieve the two literals is mutually exclusive. For example, $Have(Cake)$ and $Eaten(Cake)$ are mutex in S_1 because the only way of achieving $Have(Cake)$, the persistence action, and the only way of achieving $Eaten(Cake)$, $Eat(Cake)$, are mutually exclusive. In S_2 the two literals are not mutex because there are new ways of achieving them , such as $Bake(Cake)$ and the persistence of $Eaten(Cake)$, are not mutex.

3.4 Planning Graphs as Heuristic Estimation

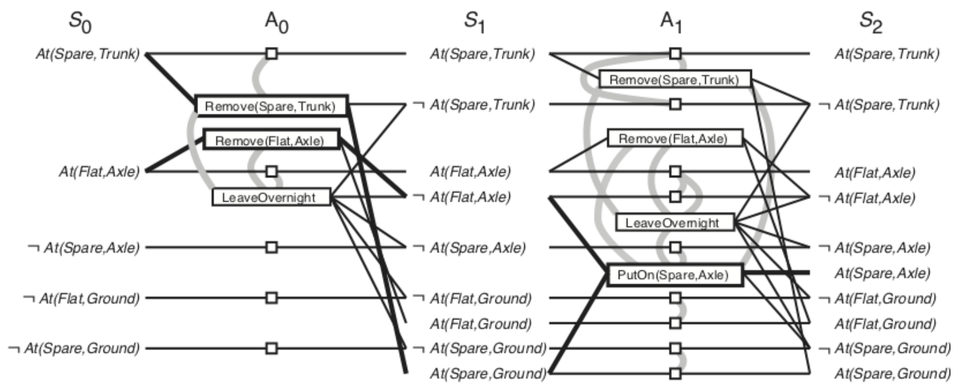
A planning graph is a reach source of information about the problem.

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

We can estimate the cost of achieving any goal literal g_i from state s as the level at which g_i first appears in the planning graph constructed from initial state s. We call this the **level cost** of g_i. The estimate might not always be accurate, however it is admissible because planning graphs allow several actions at each level, whereas heuristic counts just the level and not the number of actions.

3.4.1 Extracting a Plan

Rather than just using the graph to provide a heuristic, a plan can be extracted directly from the planning graph.



The search problem is as follow:

1) The initial state is the last level of the planning graph, S_n, along with the set of goals from the planning problem.

2) For a state at level S_i, select any conflict-free subset of the actions in A_{i-1} whose effects cover the goals in the state (Actions such that no two of them are mutex and no two of their preconditions are mutex). The resulting state has level S_{i-1} and the goal now becomes the preconditions for the selected set of actions.

3) Reach a state a level S_0 such that all the goals are satisfied.

4) The cost of each action is 1

Using the above planning graph as example. We start at S_2 with the goal $At(Spare, Axle)$. We select a set of action in A_1 such that it covers the goal, $PutOn(Spare, Axle)$.

This brings us to state S_1, and the goals become the preconditions of $PutOn(Spare, Axle)$, $\neg At(Flat, Axle)$ and $At(Spare, Ground)$. We then select any conflict free subset of actions in A_0 whose effects covers the new goal. The only solution is to choose $Remove(Spare, Trunk)$ and $Remove(Flat, Axle)$ because $LeaveOvernight$ and $Remove(Spare, Trunk)$ are mutex.

This brings us to state S_0, and the goals become the preconditions of the selected set of actions, $At(Spare, Trunk)$ and $At(Flat, Axle)$. Since both of these are present in the state, we found a solution: the actions $Remove(Spare, Trunk)$ and $Remove(Flat, Axle)$ in level A_0, followed by $PutOn(Spare, Axle)$ in A_1.

3.5 Event Calculus

The EC language is based on an ontology consisting of:

1. a set of **time-point**, which is isomorphic to the non-negative integers
2. a set of time-varying properties, called **fluents**
3. a set of **events**

Every Event Calculus description includes two main theories: **domain-independent** and **domain-dependent**

3.5.1 Domain-Independent Axioms

A set of rules that describe general principles for deciding when fluents hold or do not hold at particular time-points. These rules are:

1. Fluents that are initially true continue to hold until events occur that terminate them.

$holdsAt(f,t2) \leftarrow initially(f), not\ clipped(0,f,t2)$

2. Fluents that have been initiated by event continue to hold until events occur that terminate them.

$holdsAt(f,t2) \leftarrow initiates(a,f,t1), happens(a,t1), t1 < t2, not\ clipped(t1,f,t2)$

3. Fluents only change status via occurrences of terminating events.

$clipped(t1,f,t) \leftarrow happens(a,t2), terminates(a,f,t2), t1 < t2, t2 < t$

3.5.2 Domain-dependent Axioms

Rules that describing the particular effects of events. This usually includes:

1. Set of rules that define the effect of the action using $initiates(e,f,t)$ and $terminates(e,f,t)$.

2. A set of initial state, defining the fluents that are initially true. $initially(f)$.

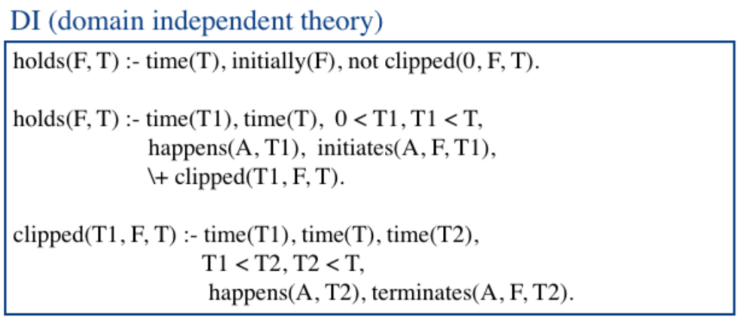
3. A set of integrity constraints

3.5.3 Abductive Planning

The knowledge base is given by the two theories of domain independent and domain dependent EC axiomatisation.

The set of abducibles is given by the set A = happens(.), i (.). The < predicates are over time variables.

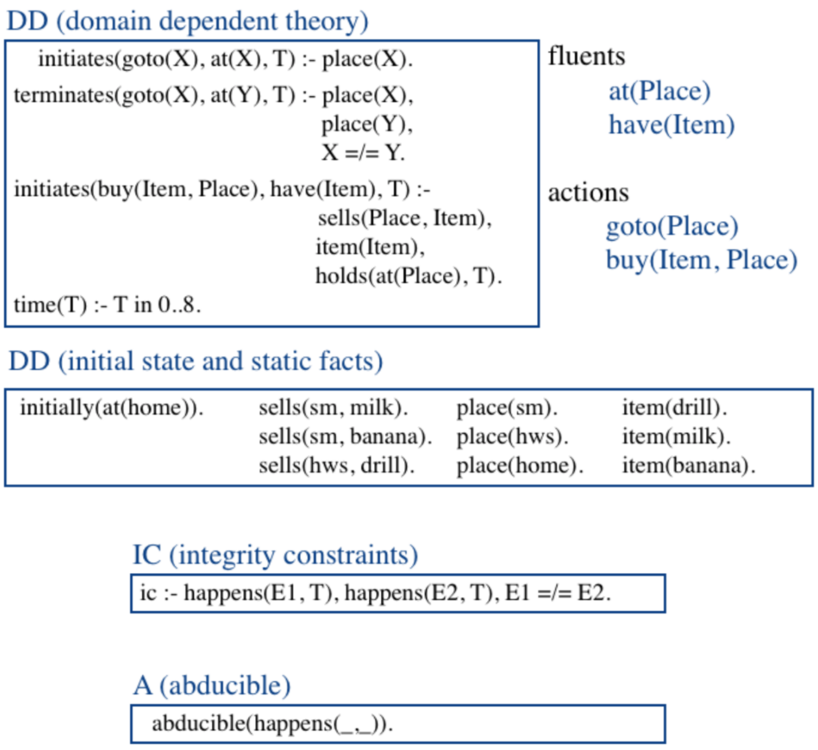
One general integrity constraint, expressing the fact that only one event can happen at each time.


DI (domain independent theory)

```
holds(F, T) :- time(T), initially(F), not clipped(0, F, T).

holds(F, T) :- time(T1), time(T),  0 < T1, T1 < T,
                 happens(A, T1),  initiates(A, F, T1),
                 \+ clipped(T1, F, T).

clipped(T1, F, T) :- time(T1), time(T), time(T2),
                        T1 < T2, T2 < T,
                         happens(A, T2), terminates(A, F, T2).
```



DD (domain dependent theory)

```
 initiates(goto(X), at(X), T) :- place(X).
terminates(goto(X), at(Y), T) :- place(X),
                                 place(Y),
                                 X =/= Y.
initiates(buy(Item, Place), have(Item), T) :-
                                 sells(Place, Item),
                                 item(Item),
                                 holds(at(Place), T).
time(T) :- T in 0..8.
```

fluents
at(Place)
have(Item)

actions
goto(Place)
buy(Item, Place)

DD (initial state and static facts)

```
initially(at(home)).    sells(sm, milk).      place(sm).      item(drill).
                        sells(sm, banana).    place(hws).     item(milk).
                        sells(hws, drill).    place(home).    item(banana).
```

IC (integrity constraints)

```
ic :- happens(E1, T), happens(E2, T), E1 =/= E2.
```

A (abducible)

```
abducible(happens(_,_)).
```

4 Logical Agents

The intelligence of humans is achieved by processes of **reasoning** that operate on internal **representations** of knowledge. In AI, this approach to intelligence is embodied in **knowledge-based agents**.

The central component of a knowledge-based agent is its **knowledge base (KB)**. A knowledge base is a set of **sentences**. Each sentence is expressed in a language called a **knowledge representation language** and represents some assertion about the world. Sometimes we dignify a sentence with the name **axiom**, when the sentence is taken as given without being derived from other sentences.

There must be a way to add new sentences to the knowledge base and a way to query what is known. **Inference** is the process of deriving new sentences from the old.

4.1 The Knowledge Base

4.1.1 Propositional Logic

We can construct a knowledge base using the semantics of **propositional logic**.

The syntax of propositional logic is given by the main logical connective $\wedge, \vee, \neg, \rightarrow$ and propositional constants.

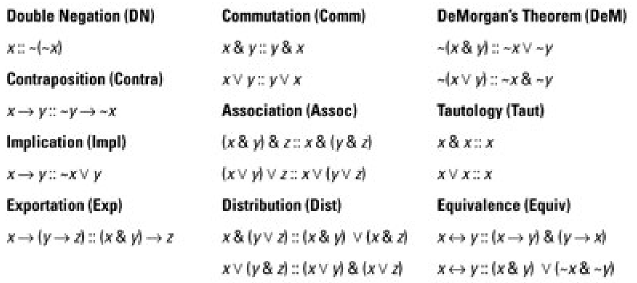
An **interpretation** that makes a given sentence true is said to **satisfy** the sentence.

A sentence is **unsatisfiable** iff it is not satisfied by any interpretation.

A sentence is **valid** iff it is satisfied by every possible interpretation.

Given a set of sentences, S, a **model** of S is an interpretation that satisfies all the sentences in S.

The relation of logical **entailment** between sentences means that one sentence follows logically from another sentence. We write $\alpha \models \beta$ to mean that sentence α entails sentence β, meaning in every model in which α is true, β is also true.



Double Negation (DN)	**Commutation (Comm)**	**DeMorgan's Theorem (DeM)**
$x :: \sim(\sim x)$	$x \,\&\, y :: y \,\&\, x$	$\sim(x \,\&\, y) :: \sim x \vee \sim y$
Contraposition (Contra)	$x \vee y :: y \vee x$	$\sim(x \vee y) :: \sim x \,\&\, \sim y$
$x \rightarrow y :: \sim y \rightarrow \sim x$	**Association (Assoc)**	**Tautology (Taut)**
Implication (Impl)	$(x \,\&\, y) \,\&\, z :: x \,\&\, (y \,\&\, z)$	$x \,\&\, x :: x$
$x \rightarrow y :: \sim x \vee y$	$(x \vee y) \vee z :: x \vee (y \vee z)$	$x \vee x :: x$
Exportation (Exp)	**Distribution (Dist)**	**Equivalence (Equiv)**
$x \rightarrow (y \rightarrow z) :: (x \,\&\, y) \rightarrow z$	$x \,\&\, (y \vee z) :: (x \,\&\, y) \vee (x \,\&\, z)$	$x \leftrightarrow y :: (x \rightarrow y) \,\&\, (y \rightarrow x)$
	$x \vee (y \,\&\, z) :: (x \vee y) \,\&\, (x \vee z)$	$x \leftrightarrow y :: (x \,\&\, y) \vee (\sim x \,\&\, \sim y)$

Every sentence in propositional logic can be expressed in **disjunctive normal form (DNF)**, e.g. $P_1 \vee P_2 \vee P_3 \vee ... \vee P_n$.

4.1.2 First Order Logic

Predicate Logic provides us with a way of talking about individual objects and their relationships. **Quantified sentences** are logical formulae formed form a quantifier, a variable and an embedded sentence.

A sentence is **ground** iff it contains no variables

The **Universal Instantiation (UI)** says that we can infer any sentence obtained by substituting a **ground term** (a term without variables) for the variable.

To write out the inference rule formally, we use the notion of **substitutions**. For σ a substitution, $\alpha\sigma$ is the formula obtained from α by applying σ. For example, the axiom $\forall x\ King(x) \rightarrow Evil(x)$ can obtain the sentences $\forall x\ King(John) \rightarrow Evil(John)$ and $\forall x\ King(Richard) \rightarrow Evil(Richard)$ with the substitutions $\{x/\text{John}\}$ and $\{x/\text{Richard}\}$.

In the rule for **Existential Instantiation**, the variable is replaced by a single new constant symbol. In logic, the new name is called the **Skolem constant**. For example, from the sentence $\exists x\ Crown(x) \wedge OnHead(x, John)$, we can infer the sentence $\exists x\ Crown(SK_1) \wedge OnHead(SK_1, John)$

Two steps are needed to transform first-order sentence into **disjunctive normal form**.

1. Eliminate existing existential quantifiers using the rules $\neg\exists x.P(x) \equiv \forall x.\neg P(x)$ and $\neg\forall x.P(x) \equiv \exists x.\neg P(x)$.

2. Remaining existentially quantified can be eliminated by introducing Skolem term.

3. Once all existential quantifiers have been removed, all universal quantifiers can be moved to front of the sentence. Then remove all universal quantifiers.

4. Change all connectives to $\vee$ by treating it like propositional logic.

4.2 First Order Inference

There are two methods to infer new sentences in first order logic.

The first idea is **propositionalisation**, just as an existentially quantified sentence can be replaced by one instantiation, a universally quantified sentence can be replaced by a the set of all possible instantiations. In other words, we reduce first-order inference to propositional inference by substituting the variables with all possible constants.

The propositionalisation approach is rather inefficient. The second approach uses a single inference rule called **Generalised Modus Ponens**. For atomic sentences p_i, p_i' and q, where there is a substitution θ such that $p_i'\theta = p_i\theta$ for all i:

$$\frac{p'_1, p'_2, \ldots, p'_n, (q \Leftarrow p_1, p_2, \ldots, p_n)}{q\theta}$$

The conclusion is the result of applying the substitution θ to the consequent q.

The second approach require finding substitutions that make different logical expressions look identical. This process is called **unification** and is a key component of all first-order inference algorithms. When there is more than one unifier to make two arguments look the same, it should return the **most general unifier (MGU)**. For example, to unify $Knows(John, x)$ and $Knowns(John, y)$ the MGU is $\{x/z, y/z\}$.

When computing most general unifiers, there is one expensive step: when matching a variable against a complex term, one must check whether the variable itself occurs inside the term; if it does the match fails because no consistent unifier can be constructed. For example $S(x)$ cannot unify with $S(S(x))$. This is called the **occur check**.

4.3 Proof by resolution

Our goal now is to decide whether the knowledge base, $KB \models \alpha$ for some sentence α.

The **resolution inference rule** takes two clauses and produces a new clause containing all the literals of the two original clauses except the two complementary literals. The removal of multiple copies of literals is called **factoring**. For example, if we resolve $(A \vee B)$ and $(A \vee \neg B)$, we obtain $(A \vee A)$ which is reduced to A. The result clause is called **resolvent clause**.

$$\frac{\alpha \vee \beta, \neg \beta \vee \gamma}{\alpha \vee \gamma} \quad \text{or equivalently} \quad \frac{\neg \alpha \rightarrow \beta, \beta \rightarrow \gamma}{\neg \alpha \rightarrow \gamma}$$

The resolution rule for **first-order** clauses is simply a lifted version of the propositional resolution rule. Propositional literals are complementary if one is the negation of the other, first-order literals are complementary if one **unifies** with the negation of the other.

For example, we can resolve the two clauses $[Animal(F(x)) \vee Loves(G(x), x)]$ and $[\neg Loves(u, v) \vee \neg Kills(u, v)]$ by eliminating the complementary literals $Loves(G(x), x)$ and $\neg Loves(u, v)$, with the unifier $\sigma = \{u/G(x), v/x\}$, to produce the resolvent clause $[Animal(F(x)) \vee \neg Kills(G(x), x)]$.

Inference procedures based on resolution work by using the principle of proof by contradiction. To show that $KB \models \alpha$, we show that $(KB \wedge \neg\alpha)$ is unsatisfiable.

First, $(KB \wedge \neg\alpha)$ is converted into CNF. Then the resolution rule is applied to the resulting clauses. Each pair that contains complementary literals is resolved to produce a new clause, which is added to the set if it is not already present. The process continues until one of the two things happens:

1) There are no new clauses that can be added, in which case KB **does not** entail α or,

2) two clauses resolve to yield the **empty clause**, in which case KB entails α.

The empty clause is equivalent to False because a disjunction is true only if at least one of its disjuncts is true.

An inference algorithm is **complete** if every true formula is provable. Resolution is complete and the completeness theorem for resolution in propositional logic is called the **ground resolution theorem**. The theorem states that if a set of clauses is unsatisfiable, then the resolution closure of those clauses contains the empty clause.

4.4 Horn Clauses and Definite Clauses

4.4.1 Propositional Logic

In many practical situations, the full power of resolution is not needed. Some real-world knowledge bases satisfy certain restrictions on the form of sentences they contain, which enables them to use a more restricted and efficient inference algorithm.

Denials are clauses with zero positive literals.

efinite clause is a disjunction of literals of which **exactly one** is positive. For example, $(\neg P_1 \vee \neg P_2 \vee P_3)$ or equivalently $(P_1 \wedge P_2 \rightarrow P_3)$.

Slightly more general is the **horn clause**, which is a disjunction of literals at which **at most one** is positive. So all definite clause are horn clauses, as are clauses with no positive literals, these are called **goal clauses**. An example of goal clause is $\neg P_1 \vee \neg P_2 \vee \neg P_3)$ or equivalently $(P_1 \wedge P_2 \wedge P_3 \rightarrow False)$.

In horn form, $(P_1 \land P_2 \land P_3 \rightarrow Q)$, the premise, $P_1 \land P_2 \land P_3$, is called the **body** and the conclusion, Q, is called the **head**. A sentence consisting of a single positive literal, Q or $(True \rightarrow Q)$, is called a **fact**.

4.4.2 First Order Definite Clauses

First-order definite clauses closely resemble propositional definite clauses: they are disjunctions of literals of which **exactly one** is positive. A definite clause is either

1) atomic, e.g. $King(John)$, or

2) an implication whose antecedent is a conjunction of positive literals and whose consequent is a single positive literal, e.g. $King(x) \land Greedy(x) \rightarrow Evil(x)$.

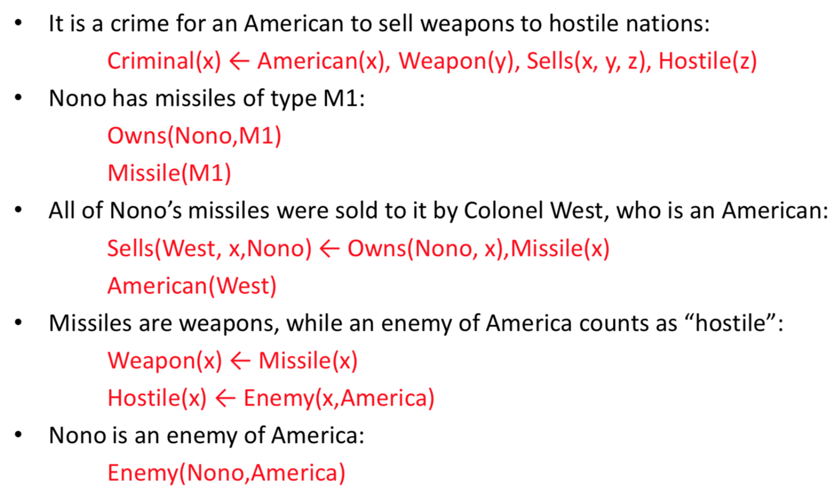
Unlike propositional literals, first-order literals can include variables, in which case those variables are assumed to be universally quantified.

Inference with Horn clauses can be done through the **forward chaining** and **backward chaining** algorithms.

4.5 Logical Systems

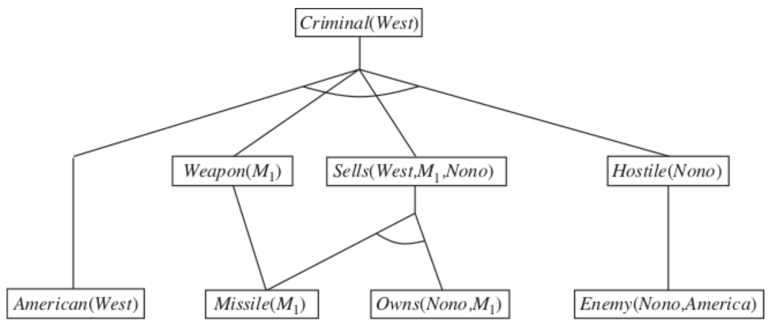
4.5.1 Forward Chaining Algorithm

Forward chaining algorithm starts from the known facts, then triggers all the rules whose premises are satisfied, adding their conclusion to the know facts. This process repeats until the query is answered, or no new facts are added.



- It is a crime for an American to sell weapons to hostile nations:
  Criminal(x) ← American(x), Weapon(y), Sells(x, y, z), Hostile(z)
- Nono has missiles of type M1:
  Owns(Nono,M1)
  Missile(M1)
- All of Nono's missiles were sold to it by Colonel West, who is an American:
  Sells(West, x,Nono) ← Owns(Nono, x),Missile(x)
  American(West)
- Missiles are weapons, while an enemy of America counts as "hostile":
  Weapon(x) ← Missile(x)
  Hostile(x) ← Enemy(x,America)
- Nono is an enemy of America:
  Enemy(Nono,America)

Suppose we are given the above set of sentences. Then the proof tree generated by forward chaining is shown below. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.



Forward chaining algorithm has three possible sources of inefficiency:

1) Consider the rule $Missile(x) \land Owns(Nono, x) \rightarrow Sells(West, x, Nono)$. We can find all the objects owned by Nono in constant time per object, then for each object check whether it is a missile. However, if the knowledge base contains many objects owned by Nono and very few missiles, it would be better to find all the missiles first then check whether they are owned by Nono. This is the **conjunct ordering** problem: to find an ordering to solve the conjuncts so that the total cost is minimised. It turns out that finding the optimal ordering is NP-hard.

2) If a rule matched the facts on iteration k then it will still match the facts on iteration k+1. Therefore there is lots of recomputation. Such redundant rule matching can be avoided if we make the following observation: Every new fact inferred on iteration k+1 must be derived from at least one new fact inferred on iteration k. This observation leads naturally to an **incremental forward-chaining** algorithm.

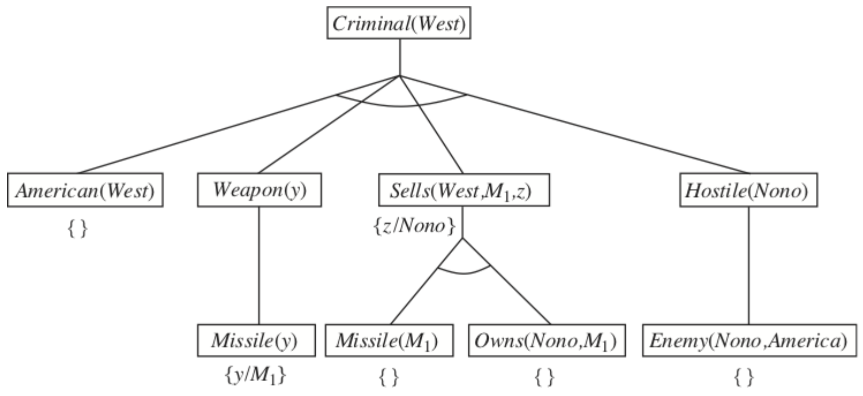
3) Forward chaining makes all allowable inferences based on the known facts, even if they are irrelevant to the goal at hand.

4.5.2 Backward Chaining Algorithm

In Backward Chaining algorithm, a goal will be proved if the knowledge base contains a clause of the form $lhs \rightarrow goal$, where lhs is a list of conjuncts. A fact such as $American(West)$ is considered as a clause whose lhs is the empty list.

Backward chaining is a kind of AND/OR search, the OR part because the goal query can be proved by any rule in the knowledge base, and the AND part because all the conjuncts in the lhs of a clause must be proved.

The proof tree generated by backward chaining is shown below. To prove $Criminal(West)$, we have to prove the four conjuncts below it. Some of these are in knowledge base, and others require further backward chaining. Bindings for each successful unification are shown next to the corresponding subgoal. Note that once one subgoal in a conjunction succeeds, its substitution is applied to subsequent subgoals. Backward chaining is a **depth-first search algorithm**



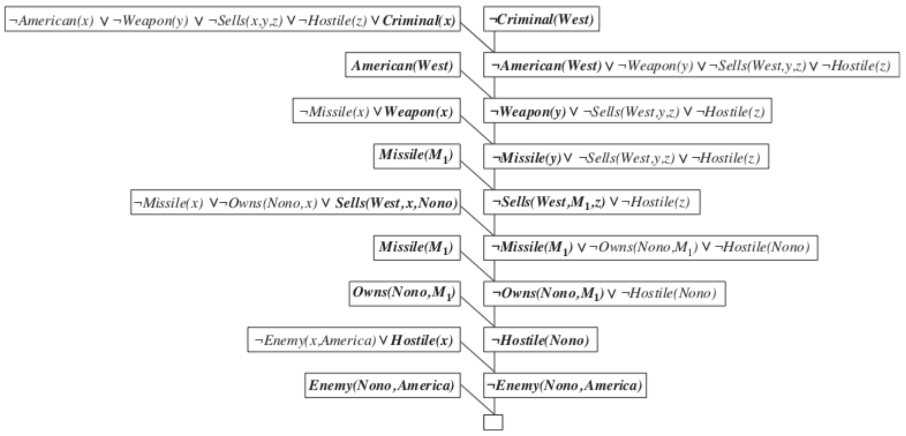
4.5.3 Resolution

Using the above example, we can use resolution proof to show that West is a criminal.

First order resolution requires that sentences to be in conjunctive normal form. Therefore we first convert all the rules to CNF.

$$\neg American(x) \lor \neg Weapon(y) \lor \neg Sells(x,y,z) \lor \neg Hostile(z) \lor Criminal(x)$$
$$\neg Missile(x) \lor \neg Owns(Nono,x) \lor Sells(West,x,Nono)$$
$$\neg Enemy(x,America) \lor Hostile(x)$$
$$\neg Missile(x) \lor Weapon(x)$$
$$Owns(Nono,M1)$$
$$American(West)$$
$$Missile(M1)$$
$$Enemy(Nono,America)$$

We also include the negated goal $\neg Criminal(West)$. The resolution proof is shown below. The spine starts with the goal clause, resolving against clauses form the knowledge base until the empty clause is generated. We always choose to resolve with a clause whose positive literal unified with the leftmost literal of the "current" clause on the spine. The kind of resolution is **Selective Linear resolution for Definite clauses (SLD resolution)**.


¬American(x) ∨ ¬Weapon(y) ∨ ¬Sells(x,y,z) ∨ ¬Hostile(z) ∨ Criminal(x)
¬Criminal(West)
American(West)
¬American(West) ∨ ¬Weapon(y) ∨ ¬Sells(West,y,z) ∨ ¬Hostile(z)
¬Missile(x) ∨ Weapon(x)
¬Weapon(y) ∨ ¬Sells(West,y,z) ∨ ¬Hostile(z)
Missile(M1)
¬Missile(y) ∨ ¬Sells(West,y,z) ∨ ¬Hostile(z)
¬Missile(x) ∨ ¬Owns(Nono,x) ∨ Sells(West,x,Nono)
¬Sells(West,M1,z) ∨ ¬Hostile(z)
Missile(M1)
¬Missile(M1) ∨ ¬Owns(Nono,M1) ∨ ¬Hostile(Nono)
Owns(Nono,M1)
¬Owns(Nono,M1) ∨ ¬Hostile(Nono)
¬Enemy(x,America) ∨ Hostile(x)
¬Hostile(Nono)
Enemy(Nono,America)
¬Enemy(Nono,America)

5 Non-Monotonic Reasoning

A definite clause logic is **monotonic** in the sense that anything that could be concluded before a clause is added can still be concluded after it is added.

$$\text{if } S \models \alpha \text{ then } S \cup X \models \alpha$$

A logic is **non-monotonic** if some conclusions can be invalidated by adding more knowledge. Non-monotonic reasoning is useful for representing defaults. A **default** is a rule that can be used unless it is overridden by an exception.

For example, a default can be ”All birds can fly”.

$$bird(x) \rightarrow flies(x)$$

However, penguins and ostriches are birds but they cannot fly. We can modify the rule in the knowledge base to capture this abnormality.

$$\begin{aligned} bird(x) &\rightarrow flies(x) \land not\ abnormal\_bird(x) \\ penguin(x) &\rightarrow bird(x) \\ penguin(x) &\rightarrow abnormal\_bird(x) \land not\ abnormal\_penguin(x) \\ ostrich(x) &\rightarrow bird(x) \\ ostrich(x) &\rightarrow abnormal\_bird(x) \land not\ abnormal\_ostrich(x) \end{aligned}$$

By adding the atom $abnormal\_bird$ in the knowledge base can prevent the default under some conditions.

5.1 Negation As Failure (NF)

A normal logic program is a set of clauses of the form:

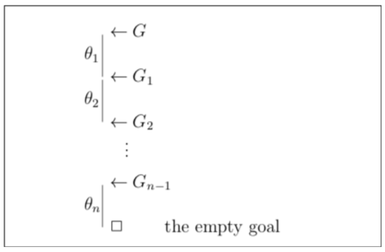
$$A \rightarrow L_1, L_2, ..., L_n \ \ (n > 0)$$

where A is an atom and each L_i is a literal. A literal is either an atom or of the form $not\ B$ where B is an atom. A clause with no occurrences of not is called a **definite clause**.

This negation, not, is often called the **negation as failure**. $not\ B$ succeeds when all attempts to prove B fail in finite time.

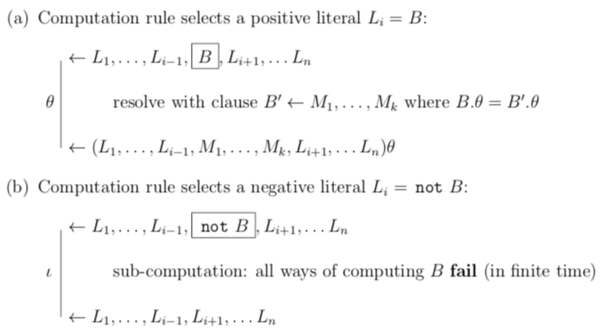
5.2 SLD and Negation As Failure (SLDNF)

In the SLDNF proof procedure, the computation of a goal/query $G = L_1, ..., L_m$ is a series of derivation steps:



The θ_i are the unifiers produced by each derivation step. The answer computed θ is the composition of these unifiers $\theta = \theta_1........\theta_n$.

In each derivation step, SLDNF chooses a sub-goal to compute. There are two kinds of derivation steps:


(a) Computation rule selects a positive literal $L_i = B$:

$\leftarrow L_1, \ldots, L_{i-1}, \boxed{B}, L_{i+1}, \ldots L_n$

θ resolve with clause $B' \leftarrow M_1, \ldots, M_k$ where $B.\theta = B'.\theta$

$\leftarrow (L_1, \ldots, L_{i-1}, M_1, \ldots, M_k, L_{i+1}, \ldots L_n)\theta$

(b) Computation rule selects a negative literal $L_i = \texttt{not}\ B$:

$\leftarrow L_1, \ldots, L_{i-1}, \boxed{\texttt{not}\ B}, L_{i+1}, \ldots L_n$

ι sub-computation: all ways of computing B **fail** (in finite time)

$\leftarrow L_1, \ldots, L_{i-1}, L_{i+1}, \ldots L_n$

When the SLDNF computes a negative literal, it just checks *not B* but does not generate bindings for variables. Therefore SLDNF must choose a **safe** subgoal to compute, meaning it must not pick a negative literal containing a variable. If the current goal contains only negative literals with variables then the computation cannot proceed: it **flounders**.

SLDNF is **non-monotonic**

5.3 Clark's Completion

Clark's completion is intended to capture the meaning of negation by failure *not* in terms of ordinary, classical, truth-functional negation $\neg$.

A database if often complete in the sense that anything not stated is false. The **complete knowledge assumption** assumes that for every atom, the clauses with the atom as the head cover all the cases when the atom is true. Under this assumption, an agent can conclude that an atom is false if it cannot that the atom is true. This is also called **closed world assumption**.

Clark's completion means that if there are not rules for an atom a, the completion of the atom is $a \leftrightarrow false$ meaning a is false.

To convert a set of sentences into Clark's completion:

1) If the language contains constant, convert it to **domain closure axioms**, and

replace all occurrences of *not* to $\neg$.

E.g. $p(X,3) \leftarrow q(X), not\ r(3)$ becomes $p(X,Y) \leftarrow Y = 3 \land q(X) \land \neg r(3)$

2) Combine all the rules with the same head using $\lor$. Add prefixing existential quantifiers for any remaining variables not in the head.

E.g. $p(X,Y) \leftarrow Y = 3 \land q(X) \land \neg r(3),\ \ p(X,Y) \leftarrow X = 2$

becomes $\forall X \forall Y(p(X,Y) \leftrightarrow (Y = 3 \land q(X) \land \neg r(3)) \lor X = 2)$

E.g. $p(X) \leftarrow X = Y \land q(Y) \land \neg r(a,Y),\ \ p(X) \leftarrow X = f(Z) \land \neg p(Z)$

becomes $\forall X(p(X) \leftrightarrow \exists Y(X = Y \land q(Y) \land \neg r(a,Y)) \lor \exists Z(X = f(Z) \land \neg p(Z)))$

3) For any predicate q of arity n appearing in the body of a clause of D but not defined in D, add the sentence $\forall x_1 ... x_n \neg q(x_1, ..., x_n)$.

5.3.1 Clark's Equality Theorem (CET)

1) Unique name axioms: $c \neq d$ for each pair of distinct constant c and d. Or more generally, $\forall x_1 ... x_n, y_1 ... y_m(f(x_1 ... x_n) \neq g(y_1 ... y_n))$ for each pair of distinct function f and g.

2) Axioms for equality: reflexivity, symmetry, transitivity, substitutivity:

$$\forall x(x = x)$$
$$\forall x \forall y(x = y \rightarrow y = x)$$
$$\forall x \forall y \forall z(x = y \land y = z \rightarrow x = z)$$
$$\forall x_1 ... x_n \forall y_1 ... y_n(x_1 = y_1 \land ... \land x_n = y_n \rightarrow p(x_1, ..., x_n) \leftrightarrow p(y_1, ..., y_n))$$

3) An axiom schema corresponding to the occurs check of the unification algorithm: $x \neq t(x)$ for any term $t(x)$ that contains x

5.3.2 Inconsistency

Consider $D = \{p \leftarrow not\ p\}$. The completion of D is $p \leftrightarrow \neg p$. We call this **inconsistent**. However, there are intuitively well-behaved and coherent logic programs/databases which arise naturally in practice but for which the Clark completion is inconsistent. Clearly, for those programs, the Clark completion is an inadequate formalisation of their intended meaning.

$$\begin{aligned}
strong(X) &\leftarrow big(X), not\ weak(X) \\
weak(X) &\leftarrow small(X), not\ strong(X), not\ abnormal\_small(X) \\
abnormal\_small(X) &\leftarrow small(X), muscular(X), not\ weak(X) \\
strong(X) &\leftarrow small(X), muscular(X), not\ weak(X)
\end{aligned}$$

Suppose that $small(Jack)$ and $muscular(Jack)$ are added to the program. We we find the completion and apply the data.

$$\begin{aligned} weak(Jack) &\leftrightarrow \neg strong(Jack) \land \neg abnormal\_small(Jack) \\ abnormal\_small(Jack) &\leftrightarrow \neg weak(Jack) \\ strong(Jack) &\leftrightarrow \neg weak(Jack) \end{aligned}$$

This suggests that the completion has two kinds of models: one in which $strong(jack)$ is true and $weak(jack)$ is false (and $abnormal\_small(jack)$ is true), and another in which $strong(jack)$ is false and $weak(jack)$ is true (and $abnormal\_small(jack)$ is false). We reach an inconsistency and so we can reach no conclusion from this program about whether jack is strong or weak.

5.3.3 Examples

Propositional programs are much simpler, because there are no quantifiers. Consider the following example D:

$$\begin{aligned} &p \leftarrow not\ q \\ &p \leftarrow r \\ &q \end{aligned}$$

Completion of D is $p \leftrightarrow (\neq q \lor r), q, \neg r$

Consider another example D:

$$\begin{aligned} p(X) &\leftarrow r(X), not\ q(X,Y) \\ p(X) &\leftarrow not\ t(X) \\ q(a,b) \end{aligned}$$

Completion of D is

$$\begin{aligned} &\forall X(p(X) \leftrightarrow \exists Y(r(X) \land \neg q(X,Y) \lor \neg t(X))), \\ &\forall X \forall Y(q(X,Y) \leftrightarrow X = a \land Y = b), \\ &\forall X \neg t(X), \\ &\forall X \neg r(X) \\ &CET \end{aligned}$$

5.4 Soundness and Completeness of SLDNF

5.4.1 Soundness of SLDNF

For normal logic program/database D, if SLDNF computes an answer substitution θ for the goal $L_1, L_2, ..., L_n$, then

$$comp(D) \models \forall((L'_1 \wedge L'_2 \wedge ... \wedge L'_n)\theta)$$

Where each L'_i is obtained from L_i by replacing all occurrences of not by $\neg$.

For normal logic program/database D, if SLDNF computation for the goal $L_1, L_2, ..., L_n$ fails finitely, then

$$comp(D) \models \forall(\neg(L'_1 \wedge L'_2 \wedge ... \wedge L'_n)) \equiv comp(D) \models \neg\exists(L'_1 \wedge L'_2 \wedge ... \wedge L'_n)$$

Where each L'_i is obtained from L_i by replacing all occurrences of not by $\neg$.

Therefore SLDNF is **sound**

5.4.2 Completeness of SLDNF

SLDNF is **not complete** because:

1) There is the possibility of floundering

2) It may go into infinite computations

Consider the following program P

$$\begin{aligned} r(a) &\leftarrow p(a) \\ r(a) &\leftarrow not\ p(a) \\ p(X) &\leftarrow p(f(X)) \end{aligned}$$

It is obvious that $P \models r(a)$ because it is true either $p(a)$ or $not\ p(a)$. However the computation of $r(a)$ will get stuck in infinite computation.

5.5 Herbrand Model

The **Herbrand universe** is the set of all ground terms obtained from constants and function symbols in S.

The **Herbrand base** is the set of all atoms from predicate symbols in S and terms in the Herbrand universe.

A **Herbrand model** is a subset of Herbrand Base that renders S true. The **Least Herbrand model** is the smallest subset of Herbrand Base that renders S true.

Example, consider the sentences $r(a,b) \rightarrow r(b,b)$, $r(a,b) \vee r(b,b)$. The Herbrand universe is $\{a,b\}$. The Hebrand Base is $\{r(a,a), r(a,b), r(b,a), r(b,b)\}$. A Herbrand Model can be $\{r(a,b), r(b,b), r(b,a)\}$. However the Least Herbrand Model is $\{r(a,b), r(b,b)\}$.

5.6 Stable Models

Stable models also called 'answer sets' provide a simple semantics for normal logic programs. The ideas of stable models can be generalised very naturally to extended logic programs, which combine negation-by-failure not with classical negation ¬.

Suppose we have a set X of atoms from the language of P. The idea is that we use X to simplify P by 'partially evaluating' all clauses with nbf-literals against X, and then we see whether the simplified program P^X we are left with (the 'reduct') has a least Herbrand model that coincides with X.

Let P be a ground normal logic program. Let X be a set of atoms. The reduct P^X is the set of clauses obtained from P as follows:

1) delete any clause in P that has a condition not A in its body where A ∈ X;

2) delete every condition of the form not A in the bodies of the remaining clauses.

E.g. $p(1,2)$, $p(X) \leftarrow p(X,Y), not\ q(Y)$

Replace these clauses by all their ground instances P:

$$\begin{aligned} &p(1,2) \leftarrow \\ &q(1) \leftarrow p(1,1), not\ q(1) \\ &q(1) \leftarrow p(1,2), not\ q(2) \\ &q(2) \leftarrow p(2,1), not\ q(1) \\ &q(2) \leftarrow p(2,2), not\ q(2) \end{aligned}$$

Let $X = \{q(2)\}$. The reduct P^X is

$$\begin{aligned} &p(1,2) \leftarrow \\ &q(1) \leftarrow p(1,1) \\ &q(2) \leftarrow p(2,1) \end{aligned}$$

The LHM $M(P^X) = \{p(1,2)\}$, so X is not stable.

Consider $X = \{p(1,2), q(1)\}$. The reduct P^X is

$$\begin{aligned} &p(1,2) \leftarrow \\ &q(1) \leftarrow p(1,2) \\ &q(2) \leftarrow p(2,2) \end{aligned}$$

The LHM $M(P^X) = \{p(1,2), q(1)\}$, so X is stable.

Every stable model of P is a minimal supported Herbrand model of P. Supported models of P are the models of $comp(P)$, the Clark's completion of P.

Stable models need not be unique, and may not exist.

6 Abductive Inference

There are three components to abductive reasoning.

1. Background knowledge which is a set of clauses

2. Abducibles which is a set of ground literals whose predicate names are not defined in the background knowledge.

3. Integrity constraints in the form of $\leftarrow A1, A2, ..., not\ B1,\ not\ B2...$ which means that it is impossible for all As to be true and all Bs to be false at the same time.

An abductive task in defined in terms of a given goal G, G is represented by a conjunction of positive and negative (NAF) literals.

An abductive explanation of an abductive task G is a set of ground abducibles that, when added to the given theory BK, explains the observation G and satisfies the integrity constraints IC.

6.1 Abductive Explanation

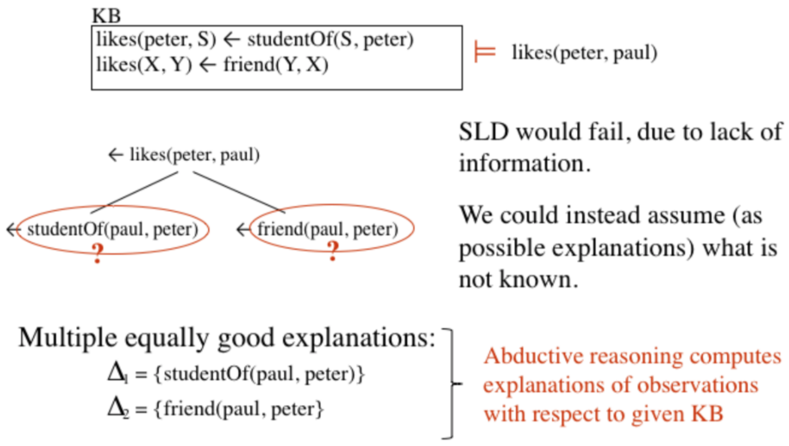
Desirable properties of abductive explanation includes: **basic**, **minimality**, **consistency**

Basic means that we cannot identify in our knowledge any further assumptions that would explain it. Meaning the abducible generated should not be the head of any rules

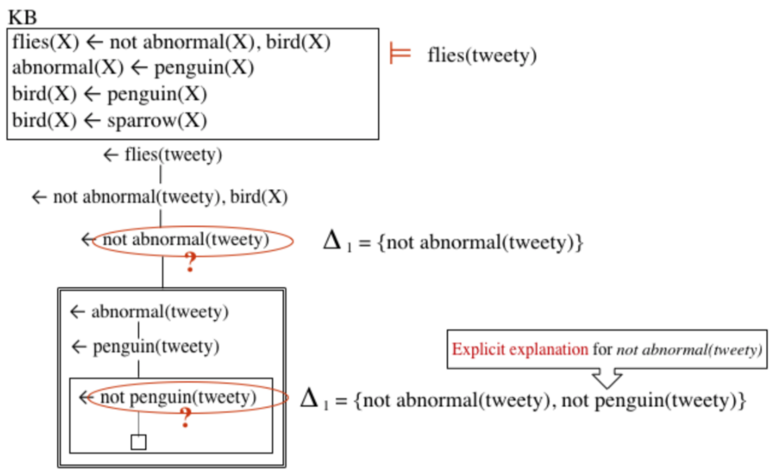
Minimality means the set of abducibles should be the minimal set. No other set of explanations should be its subset.

Consistency means the the set of explanations should take into account eh integrity constraints in the given theory.

6.2 SLD Abduction


KB
likes(peter, S) ← studentOf(S, peter)
likes(X, Y) ← friend(Y, X)
⊨ likes(peter, paul)
← likes(peter, paul)
← studentOf(paul, peter)
?
← friend(paul, peter)
?
SLD would fail, due to lack of information.
We could instead assume (as possible explanations) what is not known.
Multiple equally good explanations:
Δ1 = {studentOf(paul, peter)}
Δ2 = {friend(paul, peter}
Abductive reasoning computes explanations of observations with respect to given KB

We can imagine to extend the proof procedure by using an accumulator $\triangle$ to which unproved sub-goals are added as assumptions.


KB
flies(X) ← not abnormal(X), bird(X)
abnormal(X) ← penguin(X)
bird(X) ← penguin(X)
bird(X) ← sparrow(X)
⊨ flies(tweety)
← flies(tweety)
← not abnormal(tweety), bird(X)
← not abnormal(tweety)
?
Δ1 = {not abnormal(tweety)}
← abnormal(tweety)
← penguin(tweety)
← not penguin(tweety)
?
Explicit explanation for not abnormal(tweety)
Δ1 = {not abnormal(tweety), not penguin(tweety)}

The standard derivation is performed and when the negated condition not abnormal(tweety) is generated as sub-goal, we assume it to be true and add it to the accumulator $\triangle_1$.

But at this point a consistency phase is started (the proof in the double lined box) that tries to fail $\leftarrow abnormal(tweety)$. This proof essentially looks for explanations, in terms of abducibles, of why $abnormal(tweety)$ fails. The derivation identifies the sub-goal $\leftarrow penguin(tweety)$, which has to fail.

So try to fail it is equivalent to try to succeed its own negated form $\leftarrow not\ penguin(tweety)$.

6.3 Integrity Constraints

Satisfying a denial clause (or integrity constraint) means proving that at least one of its literals is false. The effect of the integrity constraint may be cutting possible abductive solutions or adding additional assumptions.

6.4 Abductive Proof Procedure

The main characteristic of this proof procedure is the interleave of two phases, called respectively **abductive phase** and **consistency phase**.

The negation of any predicate that appears in BK has to be added to the set of abducibles and denials of the form $\leftarrow P(X), notP(X)$ has to be implicitly added to the set of ICs.

Note that all negated literals are considered to be abducibles.

6.4.1 Abductive Phase

This phase can be called at any point of the proof. So the current (goal) literal to prove could be of different types:

1. The current goal is **not an abducible**, then standard resolution is applied.

2. The current goal is **an abducible already assumed**, then proceeds with the rest of the pending sub-goals

3. The current goal is **an abducible not yet assumed**, then the literal is assumed. Triggers the consistency phase. If such consistency derivation succeeds, then the abductive derivation can continue with the remaining pending sub-goals.

6.4.2 Consistency Phase

In the consistency phase, the new assumption is added to the current set ? and resolved with any constraint that includes it. A literal is selected from the first resolvent denial:

1. The selected literal is **not an abducible**, then standard resolution is applied.

2. The selected literal is **an abducible already assumed**, check remaining literals for failure.

3. The selected literal is **an abducible whose negation is already assumed**, the selected literal fails, and the denial is considered to be satisfied. Check next integrity rule.

4. The selected literal is **an abducible who is not assumed and its negation is also not assumed**. An abductive derivation of its negation is started. If this abductive derivation succeeds then the chosen literal fails. Check next integrity rule

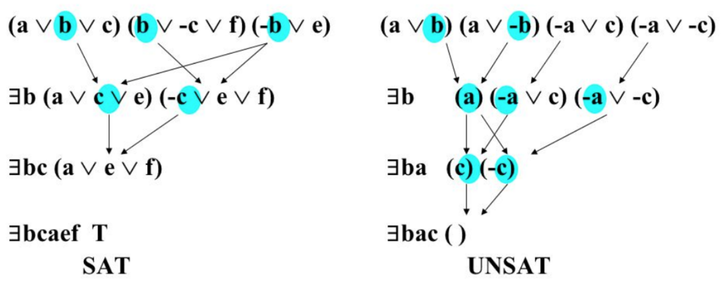
7 SAT Problem

An SAT problem is the problem of deciding if there is an assignment to the variables of a Boolean formula φ such that the formula is satisfied.

Most algorithms for the Boolean SAT problem use a **conjunctive normal form (CNF)** representation of the Boolean formula. CNF is used because it is easier to detect conflicts among clauses and in within clauses.

7.1 Davis-Putman(DP) algorithm

The simplest notion of the Davis Putman algorithm is based on **resolution**. The output is either an empty clause, it which case the set of clauses is **unsatisfiable**. Or the output stops with a set of variables, these variables will need to be satisfiable in the final assignment that is constructed. We say that these variable inform a **partial assignment**.


(a ∨ b ∨ c) (b ∨ -c ∨ f) (-b ∨ e)
∃b (a ∨ c ∨ e) (-c ∨ e ∨ f)
∃bc (a ∨ e ∨ f)
∃bcaef T
SAT
(a ∨ b) (a ∨ -b) (-a ∨ c) (-a ∨ -c)
∃b (a) (-a ∨ c) (-a ∨ -c)
∃ba (c) (-c)
∃bac ( )
UNSAT

Consider an example where $S = \{\{p, \neg q\}, \{q\}\}$. DP algorithm would return the list [p] of variables (left over) and the answer Satisfiable, stating that the given set S is satisfiable provided that p is true.

We could also eliminate all the clauses that include p. $S = \{\{q\}\}$. So we add q to the list. Therefore, [p,q] must be true to satisfy S.

7.2 Improved Davis-Putman algorithm

A **pure literal** is a literal that occurs only with one polarity.

When checking the satisfiability of S, we can safely remove all those clauses that include pure literals. For instance we can simplify the set $S = \{\{\neg x1, x2\}, \{x3, \neg x2\}, \{x4, \neg x5\}, \{x5, \neg x4\}\}$ to new set $S' = \{\{x4, \neg x5\}, \{x5, \neg x4\}\}$. Then we save in a list to remember all pure literals must be satisfied.

The **unit clause** is a clause that has one single literal l. There are two steps to the **unit propagation rules**:

1. every clause, other than the unit clause itself, in the given set S that contains the literal l is removed

2. In every clause, in the given set S, that contains $\neg l$, this negated literal is deleted.

The notion of unit clause applies also to clauses whose literals are all assigned except for one. When all literals in a clause are false under the partial assignment expect for one, this clause can be treated as a unit clause.

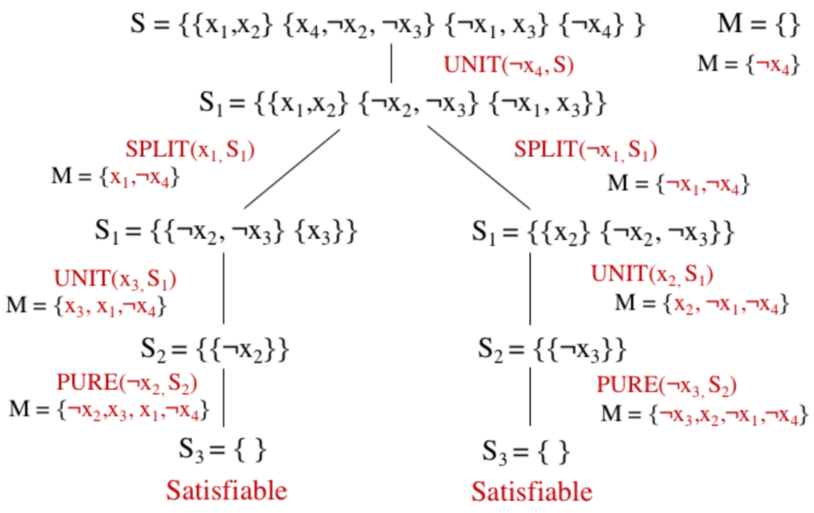
After the unit propagation rules are applied, the literal in question will clearly result to be a pure literal. Then the pure literal rule can be applied.

A **tautology clause** is a clause that includes $(p \vee \neg p)$. Tautologies can be removed since it is always satisfied.

7.3 DPLL Algorithm

The key difference between DP and DPLL is that instead of eliminating one variable at each iteration (by applying resolution), we keep the variable but we split on the two possible vales that this variable might have in the final satisfiability assignment.

The same mechanisms of pure literal rule and unit propagation are applied in DLL as in DP.



8 Answer Set Programming

In Answer Set Programming, programs are set of rules of the form $h : -b_1, b_2, ..., not\ c_1, not\ c_2 ...$ where h, b and c are made of predicates, functions, constants and variables.

A **fact** is a rule with an empty body.

head(R) is the head of a rule. $body^+(R)$ is the set of all positive body literals in a rule. $body^-(R)$ is the set of all negative body literals in a rule

8.1 Grounding

The first step of solving an Answer Set Program is to transform it into an equivalent ground program. This process is called **grounding**.

Any ground rule that contains a positive body literal that is not in the head of some other rule is **irrelevant**.

In order for grounding to be possible, all rules must be **safe**. A rule R is said to be unsafe if it contains a variable that does not occur in any positive body literal of R.

8.2 Reduct

Given a ground logic program P and an interpretation X, the reduct P^X is constructed in two steps:

1. Remove any rule whose body contains the negation of an atom in X
2. Delete any negation from the remaining rules

8.3 Minimal Model

A definite logic program P, has a unique minimal model M(P).

The minimal model of any definite logic program can be computed by the following procedures:

1. starting with M as the empty set
2. Iteratively add to M any atom that is the head of a rule in P whose body is already satisfied by M

8.4 Answer Set

An interpretation X is an answer set of a program P if and only if X is the unique minimal model of $RG(P)^X$

8.5 Choice Rule

A choice rule is in the form of $lb\{h_1;...;h_m\}ub :- b_1,...,b_n.$ where lb and ub are integers, $h_1,...,h_m$are atoms and $b_1,...,b_n$ are literals.

The head of the rule is satisfied by an interpretation X iff $lb \leq |X \cap \{h_1...h_m\}| \leq ub$.

For example, the choice rule $1\{head, tail\}1$ means that $\emptyset$ and $\{head, tail\}$ does not satisfy the rule. But the sets $\{head\}, \{tail\}$ does.

8.5.1 Reduct

When constructing the **reduct**, we must give special consideration to choice rules.

For a choice rule R, whose body is satisfied by an interpretation X:

1. If X satisfies the head of R, then for each atom h in the head, P^X contains $h :- body^+(R)$
2. If X does not satisfy the head of R, then P^X contains $\perp$: $-body^+(R)$.

The answer set of for programs with choice rules are **not the minimal model**.

8.6 Hard Constraints

A hard constraint is of the form $\bot: -b_1 ... b_n$. The constraint eliminates any answer set that satisfies $b_1 \wedge ... \wedge b_n$.

8.7 Answer Set Programming in Abduction

- $\Delta \subseteq Ab$
- $KB \cup \Delta \models G$
- $KB \cup \Delta \not\models \bot$
- $KB \cup \Delta \models IC$

```
%KB :
headache(X) :- overwork(X).
headache(X) :- migrane(X).
migrane(X) :- wrongdiet(X).
migrane(X) :- jetlag(X).
student(jane).

%IC :
 :- student(X), overwork(X).

%Ab :
overwork(jane), wrongdiet(jane), jetlag(jane)

%G :
headache(jane)
```

- $\Delta \subseteq Ab$
- $KB \cup \Delta \models G$
- $KB \cup \Delta \not\models \bot$
- $KB \cup \Delta \models IC$

```
%KB :
headache(X) :- overwork(X).
headache(X) :- migrane(X).
migrane(X) :- wrongdiet(X).
migrane(X) :- jetlag(X).
student(jane).

%IC :
 :- student(X), overwork(X).

%Ab :
0{overwork(jane);wrongdiet(jane);jetlag(jane)}3.

%G :
 :- not headache(jane).
```

An abductive task can be expressed in ASP by :

1. rewriting the abducibles as a choice rule.
2. rewriting the goal as a constraint

8.8 Brave and Cautious Semantics

An ASP program P is said to **bravely** entail an atom a if there is at least one answer set of P that contains a.

An ASP program P is said to **cautiously** entail an atom a if every answer set of P contains a.

8.9 Aggregation

Aggregation allows us to perform functions such as counts or sums.

An **aggregate element** is of the form $t_1, ..., t_j : c_1, ..., c_k$

An **aggregate** is of the form $lb\#agg\{e1; ...; en\}ub$ where $\#agg$ is the aggregate function and each e is an aggregate element

For an interpretation X and a set of ground aggregate elements E:

$$eval(X, E) = \left\{ (t_1, \ldots, t_j) \middle| \begin{array}{c} t_1, \ldots, t_j : c_1, \ldots, c_k \in E, \\ X \text{ satisfies } c_1, \ldots, c_k \end{array} \right\}$$

Aggregation functions map the set of tuples computed by the eval function to an integer.

The counting aggregate, $\#count$, simply counts the number of tuples computed by the eval function.

The $\#sum$ function is a weighted count that sums the first argument of the tuples computed by eval.

An aggregate atom has an upper and lower bound. The atom is satisfied by an interpretation if the integer computed by the aggregation function is between the bounds.

8.9.1 Grounding

Variables that only occur in an aggregate and not the rest of the rule are called **local**.

Local variables are ground inside the aggregate, and produce multiple aggregate elements, rather than multiple rules

```
person(p1). person(p2).  item(a). item(b).  cost(a,1). cost(b,2).
0{buy(P,I)}1 :- person(P),item(I).
  :- 2#sum{C,I : buy(P,I),cost(I,C)},person(P).
```

Ground Global variables

```
person(p1). person(p2).  item(a). item(b).  cost(a,1). cost(b,2).
0{buy(p1,a)}1 :- person(p1),item(a).
0{buy(p1,b)}1 :- person(p1),item(b).
0{buy(p2,a)}1 :- person(p2),item(a).
0{buy(p2,b)}1 :- person(p2),item(b).
  :- 2#sum{C,I : buy(p1,I),cost(I,C)},person(p1).
  :- 2#sum{C,I : buy(p2,I),cost(I,C)},person(p2).
```

Ground local variables inside aggregate

```
person(p1). person(p2).  item(a). item(b).  cost(a, 1). cost(b, 2).
0{buy(p1, a)}1 :- person(p1), item(a).
0{buy(p1, b)}1 :- person(p1), item(b).
0{buy(p2, a)}1 :- person(p2), item(a).
0{buy(p2, b)}1 :- person(p2), item(b).
 :- 2#sum{1, a : buy(p1, a), cost(1, a);
          2, b : buy(p1, b), cost(2, b)}, person(p1).
 :- 2#sum{1, a : buy(p2, a), cost(1, a);
          2, b : buy(p1, b), cost(2, b)}, person(p2).
```

8.9.2 Reduct

It is more standard in the ASP literature to use the FLP reduct for aggregate functions

$P^X_{FLP} = \{r \in P | X \models body(r)\}$

Here show an example with $X = \{p\}$

P:	P^X:	P^X_{FLP}:

`p :- not q.`
`q :- not p.`
	`p.`	`p :- not q.`

8.10 Weak Constraints

In many applications, some solutions are preferred to others. Weak constraints can be used to represent these preferences in ASP

A weak constraints is in the form :~ $b_1, ..., b_m[wt@lev, t_1, ..., t_n]$

The score at priority level lev if the sum of wt for all the tails $[wt@lev, t_1, ..., t_n]$ of X.

X_1 is preferred to X_2 if for the highest level lev such that score of $X_1 < X_2$

```
                ...
:~ path(X, Y), toll_cost(X, Y, TC).[TC@2, X, Y]
:~ path(X, Y), distance(X, Y, Distance).[Distance@1, X, Y]
```

In the above example, minimum cost (priority level 2) is preferred over minimum distance (priority level 1)