Technical University of Denmark

02180 Introduction to Artificial Intelligence notes

Tutor/Professor: Thomas Bolander, Nina Gierasimczuk

Teaching Assistant: _____

Group members: Emil Damsbo(s164395)

May 22, 2019

_____ _____

Emil Damsbo

Contents

Introduction

This document contains notes for the entire syllabus of the course *02180 Introduction to Artificial Intelligence* at the Technical University of Denmark. The content is up-to-date as of May 22, 2019. The course materials used for this document were distributed during the spring semester of 2019.

Defining search problems

Formal search problem notation:

1. s_0 is the initial state
2. `ACTIONS(s)`: Is the set of applicable(i.e. legal) actions in the state s.
3. `RESULTS(s,a)`: Returns the state s' reached by executing action a in s.
4. `GOAL-TEST(s)`: Returns whether or not s is a goal state.
5. `STEP-COST(s,a)`: The cost of executing a in s. Often we assume that the cost is 1 for all combinations of s and a.

We split search problems into two types: Tree-search and graph-search. The general algorithms for both follows, red lines marks the differences between each:

```
function TREE-SEARCH (problem) returns a solution, or failure
  frontier := {s0} (initial state)        // we initialise the frontier
  loop do
    if frontier = ∅ then return failure
    choose a node n from frontier
    remove n from frontier
    if n is a goal state then return solution
    for each child m of n        // we expand n
      add child m to frontier
```

```
function GRAPH-SEARCH (problem) returns a solution, or failure
  expandedNodes := ∅
  frontier := {s0} (initial state)        // we initialise the frontier
  loop do
    if frontier = ∅ then return failure
    choose a node n from frontier
    remove n from frontier
    add n to expandedNodes
    if n is a goal state then return solution
    for each child m of n        // we expand n
      if m ∉ frontier and m ∉ expandedNodes then
        add child m to frontier
```

The only difference between tree and graph search is that **graph search keeps track of repeated states**. We introduce the concept of *expanding notes*, *generated notes*, and *the frontier*:

- For tree and graph search we always generate **all** children of a state. This is called **expanding** the state, denoted by computing `RESULTS(s,a)` for all applicable actions a.
- States are called nodes. Tree search doesn't keep track of repeated states, so distinct tree-search nodes may represent the same state.
- Nodes may be of two types: **Expanded nodes** and **frontier nodes**.
- *Expanded nodes* are nodes for which all children have been generated
- *Frontier* nodes are nodes which are generated but its children haven't been computed yet.

Furthermore we have the following notions of optimality and admissibility:

- **Optimal cost of node** n is the minimal cost to achieve a goal from n as denoted by $h*(n)$.
- **Admissible heuristics** is a heuristic h where the cost of reaching a goal is never overestimated, formally $h(n) \leq h*(n)$ for all nodes n.
- **Optimal search algorithms** are algorithms that always return an optimal solution, i.e. of minimal cost.
- **A* Tree-Search** is optimal when the heuristic is admissible (does **not** hold for **A* Graph-Search**).

Uninformed Search

Also called a *blind search*. Refers to search strategies which have no additional information beyond what is provided in the problem statement. They can simply generate successors and **distinguish goal states from non-goal states**.

A search strategy is distinguished by the *order* in which nodes are expanded. We have these types of uninformed search strategies:
Breadth-first search (BFS) uses a FIFO-queue, so picking a node from the frontier dequeues it and adding a child to the frontier enqueues it. (Page 81-83 in the book)
Depth-first search (DFS) uses a LIFO-stack, so picking a node pops it from the stack and adding a child to the frontier pushes it to the stack. (Page 85-87 in the book)

Uniform-cost search uses the notion of path cost $g(n)$ and chooses the path with lowest total cost. This is done using a priority queue ordered by the path cost g.(Page 83-85 in the book)

There are many other uninformed search algorithms, but these are the most normally used. Below is a comparison of the time and space complexity for uninformed search algorithm along with whether they are complete and/or optimal:

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

Informed Search

Informed search or *heuristic search* can measure what non-goal states are "more promising", which sets them apart from uninformed search strategies. Remember that we simply differentiate by how nodes are chosen and added to the frontier. Here are some examples of informed search strategies:

Best-first search uses a priority queue based on the key $f(n)$ of a node n. It chooses a node n by extracting it from the priority queue ordered by $f(n)$. Adding a child m to the frontier adds it to the priority queue. We can choose the key $f(n)$ to be difference values. In the following, $g(n)$ denotes the cost of reaching node n from the initial state and $h(n)$ denotes some heuristic value, i.e. the estimated cost of reaching any goal from n. Depending on the combination of these functions, we gain different best-first search strategies:

- **Greedy best-first search:** $f(n) = h(n)$
- **A*:** $f(n) = g(n) + h(n)$
- **WA*** (weighted A*): $f(n) = g(n) + W * h(n)$ for some $W > 1$
- **Dijkstra:** $f(n) = g(n)$

Designing good heuristic functions $h(N)$ is important. It needs to be as precise as possible while being inexpensive to calculate (i.e. cheaper than the actual cost).

Consistent heuristics are always admissible. The step-cost $c(n, a, m)$ denotes arriving in m by executing a in n. We use this to define a consistent heuristic as $h(n) \leq c(n, a, m) + h(m)$. Remember that **A* tree-search** was optimal when h was admissible, but **A* graph-search** was not. If a heuristic h is consistent, then **A* Graph-Search** *is optimal*.

Dominating heuristics: Heuristic h_2 dominates h_1 if $h_2 \geq h_1$ for all s. If h_2 dominates h_1 as both are admissible, A^* will never expand more nodes using h_2 than using h_1. Using any combination of $h_1(s), h_2(s), \cdots, h_n(s)$ is always better than using a single h_i, unless the computation time of that combination is too high.

Non-determinism and Partial Observability

Here we generalize problem-solving by using search techniques. Previosuly we have made the following assumptions:

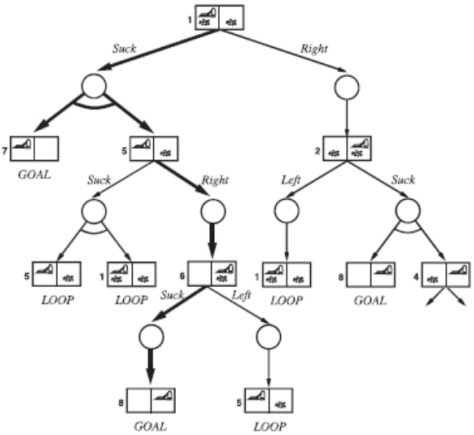
- The problem has **a single agent**, the one we control.

- The problem is **static**, the world does not change when our agent does not act.
- All actions are **deterministic**, all action outcomes are unique
- The state of our problem is **fully observable** for the agent.

Now we introduce the notion of non-deterministic actions. Here, we don't know the precise outcome of an action, only the possible outcomes. This means that `RESULTS(s,a)`now returns a set of states rather than a single state. To represent this in state trees, we use AND-OR trees:

A **solution** to a nondeterministic search problem is a subtree T' of the AND-OR tree T satisfying:

1. The root note of T is in T',
2. Every leaf of T' is a goal state,
3. Every OR node of T' has exactly one outgoing edge (agent choice),
4. Every AND node of T' has the same outgoing edges as in T (nature choice),



Example. The subtree of boldface edges is a solution.

For normal deterministic search problems we use a **sequential plan**: A sequence of actions. For non-deterministic search problems we need to use conditional plans or **contingency plans**. We use the following language for conditional plans: $\pi ::= \epsilon |$ if s then π_1 else $\pi_2 | \pi_1; \pi_2$ where ϵ is the empty plan, s is a state and $\pi_1; \pi_2$ denotes sequential composition of π_1 and π_2 (first execute π_1 then execute π_2).
We can also express solutions as **policies** where we map from states to actions, e.g. $\Pi(s_1) = a_1, \Pi(s_2) = a_2, \Pi(s_3) = a_1$.

Combining the above yields an algorithm that may return a conditional plan:

```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or
failure
  OR-SEARCH(init state of problem, [])      // problem is implicit parameter

function OR-SEARCH(state, path)
  if state is a goal then return ε      // if in goal state, empty plan suffices
  if state is on path then return failure      // fail if looping
  for each action applicable in state do      // recursively search for plan
    plan ← AND-SEARCH(RESULTS(state, action), [state | path])
    if plan ≠ failure then return action; plan      // append plan to action
  return failure      // if all recursive searches for a plan fails

function AND-SEARCH(states, path)
  for each s_i in states do      // recursively find plans for each outcome state
    plan_i ← OR-SEARCH(s_i, path)
    if plan_i = failure then return failure
  return if s_1 then plan_1 else if s_2 then plan_2 else ··· if s_{n-1} then plan_{n-1} else
         plan_n      // if n = 1 then the returned plan is just plan_1
```

Note that if all actions are deterministic the above algorithm is simplified to a recursive version of depth-first graph-search.
If a problem may cause cyclic behaviour, we need to expand our previous definition of solutions: Subtrees T' where every leaf is either a goal state or a loop node. Assuming **fairness** guarantees that cyclic solutions reach the goal. Cyclic solutions are expressed as before but with the introduction of while-conditionals: $\pi ::= \epsilon |$ if s then π_1 else $\pi_2 | \pi_1; \pi_2 |$ while *cond* do π_1.

Observability refers to what we know about the current state:
Full observability: everything about the state is known.
Partial observability: Some parts of the state is know, but not everything.
Null observability: Nothing is known about the current state. Problems of this visibility are called **conformant problems** or **sensorless problems**.

Belief states

... are a set of (physical) states which contain states considered possible by an agent in a given state. We use s to denote physical states and b to denote belief states. If a problem has N states there are up to 2^N belief states. Conforment problems can still be solved by standard graph or tree search algorithms using belief states instead of physical states.
We can convert a fully observable problem to a conformant problem. Assuming the fully observable problem (s_0, `ACTIONS`, `RESULTS`, `GOAL-TEST`) we can define the conformant problem as

$(b_0, \texttt{ACTIONS}', \texttt{RESULTS}', \texttt{GOAL-TEST}')$ by the following:

$$\begin{aligned} ACTIONS'(b) &= \cup_{s \in b} ACTIONS'(s) \\ RESULTS'(b, a) &= \cup_{s \in b} RESULTS'(s, a) \\ GOAL - TEST'(b) &= \wedge s \in b GOAL - TEST'(s) \end{aligned} \tag{1}$$

Not all conformant problems can be solved without knowing the initial state.

Adverserial Search

We take "adverserial" to mean a problem wherein two agents are trying to achieve opposite goals. This means they are each others' adversaries. In this, we assume control of one agent and try to optimize its solution to the problem.

In this section we use the following notation:

In Chapter 3 of R&N, search problems were defined by

- **Initial state**,
- ACTIONS(s): **possible actions**,
- RESULT(s, a): **transition model**,
- **Goal test**,

This induces a **state space** in which we can search for a state satisfying the goal test.

Similarly, games can be described by:

- s_0: **initial state**,
- PLAYER(s): who has the move in s,
- ACTIONS(s): Legal moves in s,
- RESULT(s, a): **transition model**,
- TERMINAL-TEST(S): **terminal test**. Is the game over?
- UTILITY(s, p): **utility function** (or **payoff function**). Numerical value for player p in terminal state s, Example: $+1$ for win and -1 for loose (zero-sum),

Minimax

One way to optimize an adversarial problem is two use a **minimax** algorithm. We call the two agents `MIN`and `MAX`in a zero-sum game. Nodes are assigned values for expected utility for `MAX`. We can then define the algorithm recursively:

$$MINIMAX(s) = \begin{cases} UTILITY(s, MAX) & \text{if } TERMINAL-TEST(s). \\ max_{a \in ACTIONS(s)} MINIMAX(RESULT(s,a)) & \text{if } PLAYER(s) = MAX \\ min_{a \in ACTIONS(s)} MINIMAX(RESULT(s,a)) & \text{if } PLAYER(s) = MIN \end{cases} \tag{2}$$

In any state s with $PLAYER(s) = MAX$, the move $\text{argmax}_{a \in ACTIONS(s)} MINIMAX(s)$ will be optimal for MAX. I.e. the player MAX chooses the move that maximises the minimax value while MIN chooses the value that minimises it.
Note that the resursive nature of minimax closely resembles that of a **depth-first search**.

Minimax is very computationally costly. We can reduce this cost using alpha-beta pruning, which reduces the tree and still guarantees optimal strategies. In this, the minimax algorithm passes down the values α and β which denotes a lower and upper bound for MAX and MIN respectively. That is, they are the so far least optimal strategies. The algorithm updates these values v iteratively. The root node has $\alpha = -\infty$ and $\beta = \infty$.

Using these values, we can define **Alpha-cut** for $v \leq \alpha$ in a MIN node and **Beta-cut** for $v \geq \beta$ in a MAX node. That is, if the condition is true then MAX and MIN, respectively, have a better choice already and we don't need to go further down the recursion. We can then define the **Alpha-beta-search** algorithm:

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

We may also need to limit the search depth to avoid spending too much computation time. To this end we may use `CUTOFF-TEST(s,d)` instead of `TERMINAL-TEST(s)` and the evaluation function `EVAL(s,p)` instead of `UTILITY(s,p)`. `EVAL(s,p)` is an estimate of the chance of "winning" or the expected utility. Expected utility is calculated using probabilities, i.e. if there is 20% chance of a utility of 5 and 80% chance of a utility of 2, then the expected utility is $0.2 \cdot 5 + 0.8 \cdot 2 = 2.6$.

Using the above we can define the **heuristic minimax** like so:

$$H - MINIMAX(s,d) = \begin{cases} EVAL(s, PLAYER(s)) & \text{if } CUTOFF - TEST(s,d). \\ max_{a \in ACTIONS(s)} H - MINIMAX(RESULT(s,a), d+1) & \text{if } PLAYER(s) = MAX \\ min_{a \in ACTIONS(s)} H - MINIMAX(RESULT(s,a), d+1) & \text{if } PLAYER(s) = MIN \end{cases} \tag{3}$$

If a problems' adversarial agent is a probabilistic decision-maker (i.e. not maximising) called `CHANCE`, then we need to define $P(s,a)$ as the probability that `CHANCE` makes the decision s in a and then we can create the **EXPECTIMAX**-algorithm:

$$EXPECTIMAX(s,d) = \begin{cases} UTILITY(s, MAX) & \text{if } TERMINAL - TEST(s,d). \\ max_{a \in ACTIONS(s)} EXPECTIMAX(RESULT(s,a)) & \text{if } PLAYER(s) = MAX \\ \Sigma_{a \in ACTIONS(s)} P(a,s) \cdot EXPECTIMAX(RESULT(s,a)) & \text{if } PLAYER(s) = CHANCE \end{cases} \tag{4}$$

Monte-Carlo tree search

This topic is really not likely to turn up on the exam, however we do want to introduce it here. Monte-Carlo tree search uses simulation to essentially play the game randomly a number of times, and then it averages the utility in each given state to denote what is *likely* to be the best choice. We use n to denote the number of times a state occurs and r to be the total utility (rewards) across all n games.

When the simulation is given you have to use minimax to find the ideal solution.

The full definition of a Monte-Carlo tree search is as follows:

Monte-Carlo Tree Search iteratively loops through the following 4 steps:

1. **Selection**: A node in the frontier is selected for expansion in the following way. We start at the root and find a path to a frontier node by iteratively selecting the best child. Upper Confidence Bound (UCB) is used to find the best child of each node on the path.
2. **Expansion**: Expand the selected node, that is, add one or more children to the node (usually only one).
3. **Simulation**: Play a random game from the generated child node. Called a **playout** or **rollout**.
4. **Backup**: Update the evaluation of all nodes on the path from the root to the generated child node based on the playout. Sometimes called **backpropagation**.

Logical Agents

Definition of validity: *An inference is **valid** where all premises are true and the conclusion is also true.* That is, if no counter-examples exist the inference is valid. Furthermore, if any premise is false nothing follows about the conclusion and also, if the conclusion is false then at least one premise is false.

Logical agents use an internal representation of a more complex real-life problem. We use logic to support this internal representation. These agents have **knowledge bases**, where sentences are stored in *knowledge representation language.* Some sentences are *axioms*, which are given to hold. In the knowledge base we define the actions TELL and ASK, which adds a sentence to the knowledge bases and queries it, respectively. Using this we can define a generic knowledge-based agent:

function KB-AGENT(*percept*) **returns** an *action*
persistent: *KB* a knowledge base, initially background knowledge;
t, a counter, initially 0 (time)

TELL(*KB*,MAKE-PERCEPT-SENTENCE(*percept*, *t*))
action ← ASK (*KB*, MAKE-ACTION-QUERY(*t*))
TELL(*KB*,MAKE-ACTION-SENTENCE(*action*, *t*))
t ← t+1
return *action*

We use the following notation:

- **Syntax**: rules for building sentences
- **Semantics**: rules determining truth of sentences
- **Model** (a possible world/intepretation): If a sentence φ is true in model m, we say that m satisfies φ and that φ is a model of φ. $M(\varphi)$ is the set of all models of φ
- **Entailment** between sentences: φ follows from ψ: $\psi \vDash$ iff $M(\psi) \subseteq M(\varphi)$
- **Model-checking**: Procedure of logical inference that enumerates all models of the knowledge base and checks if the conclusion holds in all those models
- If a logical inference algorithm can *derive* a sentence φ from the knowledge base KB then we use the notation $KB \vdash \varphi$

- **Semantic model-checking**: Enumerating models and checking that a proof holds in all models.
- **Syntactic model-checking**: Applying rules of inference to the sentences in our knowledge base to construct a proof without consulting any models.

Inference in Propositional Logic

Definite clauses are clauses of literals in which exactly one is positive.
A Horn clause is a disjunction of literals where *at most one* is positive. All definite clauses are Horn clauses. Horn clauses are closed under resolution and resolving two horn clauses yields another Horn clause.
Inference with Horn clauses can be done using **backward- and forward chaining**.
A Formula is **satisfiable** if some assignment of truth values causes the formula to be true.
If a value is true for all truth value assignment, the formula is **valid** and a **tautology**.
We define CNF (Conjunctive Normal Form) as a formula that contains only conjunctions and disjunctions. This is used in the resolution algorithm (see logic notes).

Checking entailment of $\varphi \vDash \psi$ is done by testing that $\varphi \wedge \neg\psi$ is unsatisfiable. We have an algorithm that improves on truth table satisfiability checking using the David Putnam algorithm:

Improvements over TT:

1. Early termination, e.g., if $(A \vee B) \wedge (A \vee C)$ is true if A is true, regardless of B and C.
2. Pure symbol heuristic, e.g., in $(A \vee \neg B)$, $(\neg B \vee \neg C)$, $(C \vee A)$, the symbol A is pure. If a sentence has a model, then it has a model with the pure symbols assigned so as to make their literals true, because doing so can never make a clause false.
3. Unit clause heuristic, e.g., if the model contains $B = \top$, then $(\neg B \vee \neg C)$ simplifies to $\neg C$, which is a unit clause. Assigning one unit clause can create another unit clause, such 'cascade' of forced assignments is called unit propagation.

function DPLL-SATISFIABLE?(s) **returns** *true* or *false*
  inputs: s, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of s
  symbols ← a list of the proposition symbols in s
  return DPLL(*clauses*, *symbols*, { })

---

function DPLL(*clauses*, *symbols*, *model*) **returns** *true* or *false*

  if every clause in *clauses* is true in *model* **then return** *true*
  if some clause in *clauses* is false in *model* **then return** *false*
  P, *value* ← FIND-PURE-SYMBOL(*symbols*, *clauses*, *model*)
  if P is non-null **then return** DPLL(*clauses*, *symbols* − P, *model* ∪ {P=*value*})
  P, *value* ← FIND-UNIT-CLAUSE(*clauses*, *model*)
  if P is non-null **then return** DPLL(*clauses*, *symbols* − P, *model* ∪ {P=*value*})
  P ← FIRST(*symbols*); *rest* ← REST(*symbols*)
  return DPLL(*clauses*, *rest*, *model* ∪ {P=*true*}) **or**
      DPLL(*clauses*, *rest*, *model* ∪ {P=*false*}))

The WalkSat algorithm picks an unsatisfied clause and picks a symbol in it to flip. It chooses randomly between two ways to pick which symbol to flip:

1. a **min-conflicts** step that minimises the number of unsatisfied clauses in the new state and
2. a **random walk** step that picks the symbol randomly

function WALKSAT(*clauses*, *p*, *max_flips*) **returns** a satisfying model or *failure*
  inputs: *clauses*, a set of clauses in propositional logic
    p, the probability of choosing to do a "random walk" move, typically around 0.5
    max_flips, number of flips allowed before giving up

  model ← a random assignment of *true*/*false* to the symbols in *clauses*
  for $i = 1$ **to** *max_flips* **do**
    if *model* satisfies *clauses* **then return** *model*
    clause ← a randomly selected clause from *clauses* that is false in *model*
    with probability *p* flip the value in *model* of a randomly selected symbol from *clause*
    else flip whichever symbol in *clause* maximizes the number of satisfied clauses
  return *failure*

Belief Revision and Learning

A belief is some sentence (in a some formal language) and the beliefs of an agent is a set of such sentences (a belief set). Typically this formal language is propositional logic using propositions $p, q, r, \cdots$ and logical connectives. These beliefs can be used to form **logical consequences**. For any set A of sentences, $Cn(A)$ is the set of logical consequences of A. This set $Cn(A)$ satisfies the following:

- $A \subseteq Cn(A)$ (inclusion)
- If $A \subseteq B$, then $Cn(A) \subseteq Cn(B)$ (monotonicity)
- $Cn(A) = Cn(Cn(A))$ (Iteration)

If φ can be derived from A by propositional logic, then $\varphi \in Cn(A)$.

There are three things we can do to a belief set. In the following we prioritize new information:

1. **Revision**: $B * \varphi$; φ is added and other things are removed such that the resulting belief set B' is consistent.
2. **Contraction**: $B \div \varphi$; φ is removed from B resulting in the new belief set B'
3. **Expansion**: $B + \varphi$; φ is added to B resulting in the new belief set B'

Note that revision is defined using the Levi identity of contraction and expansion: $B * \varphi := (B \div \neg\varphi) + \varphi$.

We want $B \div \varphi$ to be a subset of B that does not imply φ, but we do not want any unnecessary removals from B. We say that for any set A and sentence φ the **remainder** for A and φ is any set C such that:

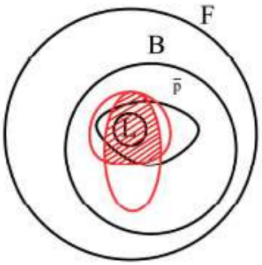
- C is a subset of A,
- C does not imply φ, and
- there is noset C' such that it does not imply φ and $C \subset C' \subseteq A$

Contraction is not deterministic, there can be different resulting sets of our belief sets depending on how it is done. Example:

$$\begin{aligned} &B = \{p, q, p \to q\} \\ &\text{New information: } \neg q \\ &\text{Remainders: } B' = \{p \to q\}, B'' = \{p\} \end{aligned} \tag{5}$$

Both are equally valid remainders. The first one decided to revise such that both p and q was false, resulting in having to remove p but keeping that $p \to q$, because false implies everything even false. The second option decided that p might still be true, but true does not imply false ergo $p \to q$ had to be contracted as well, leaving us with the remainder p.

Partial meet contraction is a solution where we let contraction $B \div \varphi$ be the **intersection** of some of the remainders. We have a **selection function** for B called γ such that if $B \perp \varphi$ is non-empty then $\gamma(B \perp \varphi)$ is a non-empty subset of $B \perp \varphi$. The output of *partial meet contraction* is equal to the intersection of the selected elements of $B \perp \varphi$ i.e. $B \div \varphi = \cap\gamma(B \perp \varphi)$.



1. Closure: $B \div \varphi = Cn(B \div \varphi)$
   the outcome is logically closed
2. Success: If $\varphi \notin Cn(\emptyset)$, then $\varphi \notin Cn(B \div \varphi)$
   the outcome does not contain φ
3. Inclusion: $B \div \varphi \subseteq B$
   the outcome is a subset of the original set
4. Vacuity: If $\varphi \notin Cn(B)$, then $B \div \varphi = B$
   if the incoming sentence is not in the original set then there is no effect
5. Extensionality: If $\varphi \leftrightarrow \psi \in Cn(\emptyset)$, then $B \div \varphi = B \div \psi$.
   the outcomes of contracting with equivalent sentences are the same
6. Recovery: $B \subseteq (B \div \varphi) + \varphi$.
   contraction leads to the loss of as few previous beliefs as possible

$\div$ is the operator of *partial meet contraction* for a belief set B if and only if it satisfies the postulates above. Using this we can also define that revision $*$ is defined via the Levi identity from earlier; $B * \varphi = (B \div \neg\varphi) + \varphi$ which has the following properties:

R1 Closure: $B * \varphi = Cn(B * \varphi)$
R2 Success: $\varphi \in B * \varphi$
R3 Inclusion: $B * \varphi \subseteq B + \varphi$
R4 Vacuity: If $\neg\varphi \notin B$, then $B * \varphi = B + \varphi$
R5 Consistency: $B * \varphi$ is consistent if φ is consistent.
R6 Extensionality: If $(\varphi \leftrightarrow \psi) \in Cn(\emptyset)$, then $B * \varphi = B * \psi$.
And if $\div$ is a transitive relational partial meet contraction, then $*$ satisfies also:
R7 Superexpansion: $B * (\varphi \wedge \psi) \subseteq (B * \varphi) + \psi$
R8 Subexpansion: If $\neg\psi \notin B * \varphi$, then $(B * \varphi) + \psi \subseteq B * (\varphi \wedge \psi)$.

Inference in First-Order Logic

First-order logic defines the following notions:

- Names for **objects**:
  - variables $x, y, z, \cdots$ when the object is indefinite
  - function symbols $a, b, c, \cdots$ for *constants*, i.e. special objects
- Properties and predicates of objects

  - Capital letters denote predicates with n arguments (n-ary)
  - 1-place predicates are intransitive verbs like "walk" or common nouns "being a boy" (you x are P a boy)
  - 2-place predicates are transitive verbs like "see" (you x see P a thing y)
  - 3-place predicates are ditransitive verbs like "give" (you x gives P an item y to another person z)
- Uses a combination of the usual logical connectives from propositional logic and quantifiers (universal $\forall$ and existential $\exists$)

$$
\begin{aligned}
\mathit{Sentence} &\rightarrow \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} &\rightarrow \mathit{Predicate} \mid \mathit{Predicate}(\mathit{Term}, \ldots) \mid \mathit{Term} = \mathit{Term} \\
\mathit{ComplexSentence} &\rightarrow (\ \mathit{Sentence}\ ) \mid [\ \mathit{Sentence}\ ] \\
&\mid \neg\ \mathit{Sentence} \\
&\mid \mathit{Sentence} \wedge \mathit{Sentence} \\
&\mid \mathit{Sentence} \vee \mathit{Sentence} \\
&\mid \mathit{Sentence} \Rightarrow \mathit{Sentence} \\
&\mid \mathit{Sentence} \Leftrightarrow \mathit{Sentence} \\
&\mid \mathit{Quantifier}\ \mathit{Variable}, \ldots\ \mathit{Sentence} \\
\\
\mathit{Term} &\rightarrow \mathit{Function}(\mathit{Term}, \ldots) \\
&\mid \mathit{Constant} \\
&\mid \mathit{Variable} \\
\\
\mathit{Quantifier} &\rightarrow \forall \mid \exists \\
\mathit{Constant} &\rightarrow A \mid X_1 \mid \mathit{John} \mid \cdots \\
\mathit{Variable} &\rightarrow a \mid x \mid s \mid \cdots \\
\mathit{Predicate} &\rightarrow \mathit{True} \mid \mathit{False} \mid \mathit{After} \mid \mathit{Loves} \mid \mathit{Raining} \mid \cdots \\
\mathit{Function} &\rightarrow \mathit{Mother} \mid \mathit{LeftLeg} \mid \cdots
\end{aligned}
$$

When working with first-order logic we need a framework for thinking in the terms of the original problem using the language of FOL. For this we use the following process:

1. Identify the task
2. Assemble the relevant knowledge about the problem
3. Decide on a vocabulary (i.e. define the terms and their meaning in FOL)

4. Encode general knowledge of the domain
5. Encode a description of the specific problem instance
6. Pose queries to the inference procedure and get some answers
7. Debug the knowledge base (i.e. find inconsistencies, redundant sentences etc.)

When inferring in FOL we need two definitions:

- **Universal Instantiation:** Infer any sentence obtained by substituting a ground term (a term without variables). This can be applied many times to produce different consequences.
- **Existential Instantiation:** Replace the variable by a single new constant symbol. This can be used once after which the existential sentence is discarded.

After applying the two instantiations the knowledge base KB is *not logically equivalent*, but it can be shown to be **inferentially equivalent**. This means that the resulting knowledge base KB' is satisfiable when KB is satisfiable.

For notes about unification, resolution, Skolemization and CNF see the notes from 02156 Logic Programming and Logical Systems.