Artificial Intelligence I
Summary

Lorenz Gorse

February 3, 2018

DISCLAIMER: THIS IS NOT A COMPLETE SUMMARY. ERRORS ARE LIKELY. THE SOLE SOURCE ARE THE LECTURE SLIDES OF THE LECTURE "KÜNSTLICHE INTELLIGENZ I" OF THE WINTER TERM 16/17 AT THE FRIEDRICH-ALEXANDER UNIVERSITY.

1 Agents

Definition 1 (Agent)**.** An agent is an entity that perceives and acts. It is modeled as a function from percept histories to actions.

$$\mathcal{P}^* \mapsto \mathcal{A}$$

An agent is anything that perceives its environment via sensors and acts on it with actuators.

Definition 2 (Performance measure)**.** A performance measure is a function that evaluates a sequence of environments.

Definition 3 (Rationality)**.** An agent is called rational, if it chooses the actions that maximizes the expected value of the performance measure given the percept history to date.

Definition 4 (Autonomy)**.** An agent is called autonomous, if it does not rely on the prior knowledge of the designer. Autonomy avoids fixed behaviors that can become unsuccessful in a changing environment.

Definition 5 (Task environment)**.** A combination of a performance measure, environment, actuators and sensors (PEAS) describes a task environment.

Definition 6 (Environment properties)**.** For an agent a we call an environment e

fully observable, iff a's sensors give it access to the complete state of e at any point in time, else **partially observable**.

deterministic, iff the next state of e is completely determined by a's action and e's current state, else **stochastic**.

episodic, iff a's experience is divided into atomic, independent episodes, where it perceives and performs a single action. Non-episodic environments are called **sequential**.

dynamic, iff e can change without an action performed by a, else **static**.

discrete, iff the sets of e's states and a's actions are countable, else **continuous**.

single-agent, iff only a acts on e.

Definition 7 (Simple reflex agent)**.** This is an agent that bases its next action only on the most recent percept.

$$f_a : \mathcal{P} \mapsto \mathcal{A}$$

Definition 8 (Reflex agent with state)**.** This is an agent with a FSM as a agent function. Inputs to the FSM are the percepts, outputs are a function of the current state and determine the actions.

Definition 9 (Goal-based agent)**.**

Definition 10 (Utility-based agent)**.** This is an agent that uses a world model along with a utility function that measures its preference among the states of the world. It chooses the action that leads to the best expected utility, which is computed by averaging over all possible outcome states, weighted by the probability of the outcome.

Definition 11 (Learning agent)**.** This is an agent that augments the performance element (which chooses actions from percept sequences) with a

learning element that makes improvements to the agent's performance element

critic which gives feedback to the learning element based on an external performance standard

problem generator which suggests actions that can lead to new, informative experiences

Definition 12 (State representation)**.** We call a state representation

atomic if it has no internal structure

factored if each state is characterized by attributes and their values

structured if the state includes objects and their relations

2 Search

Definition 13 (Search problem)**.** A search problem

$$\mathcal{P} = \langle \mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$$

consists of a set $\mathcal{S}$ of states and a set $\mathcal{O} \subseteq \mathcal{S} \times \mathcal{S}$ of operators. Certain states in $\mathcal{S}$ are labeled as goal states $\mathcal{G} \subseteq \mathcal{S}$ and there is a single initial state $\mathcal{I} \in \mathcal{S}$. A cost function $c : \mathcal{O} \mapsto \mathbb{R}^+$ assigns costs to operators.

Definition 14 (Solution)**.** A sequence of operators that lead from the initial state $\mathcal{I}$ to any goal state $g \in \mathcal{G}$ is a solution.

Definition 15 (Problem types)**.** Problems come in many variations:

Single-state problem : state is always known with certainty (observable, deterministic, static, discrete)

Multiple-state problem : know which states might be in (initial state not/partially observable)

Contingency problem : constructed plans with conditional parts based on sensors (non-deterministic, unknown state space)

Definition 16 (Tree search)**.** This is an algorithm that explores state spaces according to a search strategy.

Definition 17 (Search strategy)**.** A search strategy orders the nodes in the fringe. The following properties of search strategies are studied:

Completeness: Does it always find a solution if one exists?

Time complexity: Number of nodes generated/expanded.

Space complexity: Maximum number of nodes held in memory.

Optimality: Does it always find the least-cost solution?

Definition 18 (Uninformed search)**.** Search strategies that only employ information from the problem statement yield uninformed searches. Examples are breadth-first-search (BFS), uniform-cost-search (UCS), depth-first-search (DFS), depth-limited search, iterative-deepening-search (IDS).

Definition 19 (Informed search)**.** Search strategies that use information about the real world beyond the problem statement yield informed searches. The additional information about the world is provided in form of heuristics. Examples are greedy-search and A*-search.

Definition 20 (Heuristic)**.** A heuristic is an evaluation function $h : \mathcal{S} \mapsto \mathbb{R}^+ \cup \{\infty\}$ that estimates the cost from a state n to the nearest goal state. If $n \in \mathcal{G}$, then $h(n) = 0$.

Definition 21 (Goal distance function)**.** The function $h^* : \mathcal{S} \mapsto \mathbb{R}^+ \cup \{\infty\}$ with $h^*(n)$ being the cost of the cheapest path from n to a goal state is the goal distance function.

Definition 22 (Admissibility and consistency)**.** A heuristic h is admissible if $h(n) \leq h^*(n)$ for all states $n \in \mathcal{S}$. A heuristic h is consistent if $h(s) - h(s') \leq c(o)$ for all $s \in \mathcal{S}$ and $o = (s, s') \in \mathcal{O}$.

Definition 23 (Greedy-search)**.** Greedy-search always expands the node that seems closest to a goal state, as measured by the heuristic.

Definition 24 (A^*-search)**.** A^*-search always expands the node that has the smallest sum of current path cost and estimated distance to the nearest goal state.
The evaluation function for A^*-search is given by $f(n) = g(n) + h(n)$, where $g(n)$ is the path cost for n and $h(n)$ is the estimated cost to goal from n.

Theorem 1. A^*-search is optimal if it uses an admissible heuristic.

Definition 25 (Dominance)**.** Let h_1 and h_2 be two admisible heuristics we say that h_2 dominates h_1 if $h_2(n) \geq h_1(n)$ for all n.

Definition 26 (Local search)**.** A search algorithm that only operates on a single space at a time is called a local search. Local search algorithms need constant space. One example is hill-climbing:

1. Start with any state n.
2. Move to the successor n of the current state for which $h(n)$ is minimal.

Definition 27 (Game state space)**.** A 6-tuple $\Theta = (S, A, T, I, S^T, u)$ is a game state space.

- S are the states of the game. This is the disjoint union of S^{Max} (When it's Max's move), S^{Min} (When it's Min's move) and S^T (When the game is over).
- A are the possible moves. This is the disjoint union of
$$A^{Max} \subseteq S^{Max} \times (S^{Min} \cup S^T)$$
and
$$A^{Min} \subseteq S^{Min} \times (S^{Max} \cup S^T)$$
- S^T is the set of terminal states.
- $u : S^T \mapsto R$ is the utility function. Each terminal state is assigned a value, which is interpreted as the score for Max (and the negative score for Min).

Definition 28 (Strategy)**.** Let Θ be a game state space, and let $X \in \{\max, \min\}$. A strategy for X is a function $\sigma^X : S^X \to A^X$ so that a is applicable to s whenever $\sigma^X(s) = a$. A strategy is optimal if it yields the best possible utility for X assuming perfect opponent play.

Definition 29 (Minimax Algorithm)**.** The minimax algorithm is given by the following function whose input is a state $s \in S^{\max}$, in which Max is to move:

function Minimax-Decision(s) **returns** an action

1: v :=Max-**Value**(s)
2: **return** an action yielding value v in the previous function call

function Max-**Value**(s) **returns** a utility value

1: **if** Terminal-Test(s) **then return** $u(s)$
2: $v := -\infty$
3: **for** each $a \in$ Actions(s) **do**
4: $\quad v := \max(v, \text{Min-}\mathbf{Value}(\text{ChildState}(s, a)))$
5: **return** v

function Min-**Value**(s) **returns** a utility value

1: **if** Terminal-Test(s) **then return** $u(s)$
2: $v := +\infty$
3: **for** each $a \in$ Actions(s) **do**
4: $\quad v := \min(v, \text{Max-}\mathbf{Value}(\text{ChildState}(s, a)))$
5: **return** v

Definition 30 (Alpha-Beta-Search)**.** The alphabeta search algorithm is given by the following pseudo-code:

function Alpha-Beta-Search(s) **returns** an action

1: $v :=$Max-**Value**$(s, -\infty, +\infty)$
2: **return** an action yielding value v in the previous **function** call

function Max-**Value**(s, α, β) **returns** a utility value

1: **if** Terminal-Test(s) **then return** $u(s)$
2: $v := -\infty$
3: **for** each $a \in$ Actions(s) **do**
4: $\quad v := \max(v, \text{Min-}\mathbf{Value}(\text{ChildState}(s, a), \alpha, \beta))$
5: $\quad \alpha := \max(\alpha, v)$
6: $\quad$ **if** $v \geq \beta$ **then return** v
7: $\quad$ /* Here: $v \geq \beta \Leftrightarrow \alpha \geq \beta$ */
8: **return** v

function Min-**Value**(s, α, β) **returns** a utility value

1: **if** Terminal-Test(s) **then return** $u(s)$
2: $v := +\infty$
3: **for** each $a \in$ Actions(s) **do**
4: $\quad v := \min(v, \text{Max-}\mathbf{Value}(\text{ChildState}(s, a), \alpha, \beta))$
5: $\quad \beta := \min(\beta, v)$
6: $\quad$ **if** $v \leq \alpha$ **then return** v
7: $\quad$ /* Here: $v \leq \beta \Leftrightarrow \alpha \geq \beta$ */
8: **return** v

Definition 31 (Monte-Carlo sampling)**.** For Monte-Carlo sampling we evaluate actions through sampling. When deciding which action to take on game state s:

function Monte-Carlo sampling

1: **while** time not up **do**
2: $\quad$ select action a applicable to s
3: $\quad$ run random sample from a until terminal state t
4: **return** an a for s with maximal average $u(t)$

Definition 32 (Monte-Carlo Tree Search)**.** For Monte-Carlo tree search we maintain a search tree T:

function Monte-Carlo Tree Search

1: **while** time not up **do**
2: $\quad$ apply actions within T to select a leaf state s'
3: $\quad$ select action a' applicable to s'
4: $\quad$ run random sample from a'
5: $\quad$ add s' to T, update averages etc.
6: **return** an a for s with maximal average $u(t)$
7: When executing a, keep the part of T below a.

3 Constraint Satisfaction Problems

Definition 33 (Constraint Satisfaction Problem)**.** Abbreviated as CSP. This is a search problem where the states are induced by a finite set $V = \{X_1, \ldots, X_n\}$ of variables and domains $\{D_v | v \in V\}$ and the goal test is a set of constraints specifying allowable combinations of values for subsets of variables.

A partial assignment is a partial function $a : V \mapsto \bigcup_{v \in V} D_v$ if $a(v) \in D_v$ for all $v \in V$. We call a an assignment if a is total.

A partial assignment a is inconsistent, iff there are variables $u, v \in V$ and a constraint $C_{uv} \in C$ and $(a(u), a(v)) \notin C_{uv}$. a is called consistent iff it is not inconsistent.

A consistent assignment is called as solution.

The states of the problem are the partial assignments, the initial state is the empty assignment, the goal states are the consistent assignments and the operators are partial assignment extensions.

The CSP is called binary if all constraint relate at most two variables.

Definition 34 (Constraint network)**.** A triple $\langle V, D, C \rangle$ is called a constraint network, where

- $V = \{X_1, \ldots, X_n\}$ is a finite set of variables
- $D = \{D_v | v \in V\}$ the set of their domains
- $C = \{C_{uv} | u, v \in V\}$ a set of binary constraints. Each binary constraint is a symmetric relation $C_{uv} \subseteq D_u \times D_v$.

Remark 1. Binary CSPs can be formulated as constraint networks.

Definition 35 (Minimum remaining values)**.** This is a heuristic for backtracking searches on CSPs. It says: Assign the variable with the fewest remaining legal values next.

Definition 36 (Degree heuristic)**.** This is a heuristic for backtracking searches on CSPs. It says: Assign the variable with the most constraints on remaining variables next.

Definition 37 (Least constraining value)**.** This is a heuristic for backtracking searches on CSPs. It says: When assigning a variable, choose the value that rules out the fewest values from the neighboring domains.

Remark 2. A popular heuristic combines the three heuristics above: From the set of most constrained variables, pick the most constraining variable and assign the least constraining value.

Definition 38 (Equivalent constraint networks)**.** Two constraint networks $\gamma = \langle V, D, C\rangle$ and $\gamma' = \langle V, D', C'\rangle$ are equivalent ($\gamma \equiv \gamma'$) iff they have the same solutions.

Definition 39 (Tightness)**.** Let $\gamma = \langle V, D, C\rangle$ and $\gamma' = \langle V, D', C'\rangle$ be two constraint networks. γ' is said to be tighter than γ ($\gamma' \sqsubseteq \gamma$) iff

- For all $v \in V$: $D'_v \subseteq D_v$
- For all $u, v \in V$: $C_{uv} \notin C$ or $C'_{uv} \subseteq C_{uv}$

Remark 3. An equivalent but tighter constraint network is preferable, because it has fewer consistent partial assignments.

Definition 40 (Backtracking with Inference)**.** The general algorithm for backtracking with inference:

function BacktrackingWithInference(γ, a) **returns** a solution, or "inconsistent"

if a is inconsistent **then return** "inconsistent"
if a is a tgtal assignment **then return** a
$\gamma' :=$ a copy of γ /* $\gamma' = (V, D', C')$ */
$\gamma' :=$ Inference(γ')
if exists v with $D'_v = \emptyset$ **then return** "inconsistent"
select some variable v for which a is not defined
for each $d \in$ copy of D'_v in some order **do**
    $a' := a \cup (v = d)$; $D'_v = \{d\}$
    /* makes a explicit as a constraint */
    $a'' :=$ BacktrackingWithInference(γ', a')
    if $a'' \neq$ "inconsistent" **then return** a''
return "inconsistent"

Remark 4. Inference(γ): Any procedure delivering a (tighter) equivalent network.

Definition 41 (Forward checking)**.** For a constraint network γ and a partial assignment a, remove all values from the domains of unassigned variables that are in conflict with the values of already assigned variables to obtain a tighter network γ'.

function ForwardChecking(γ, a) **returns** modified γ

for each v where $a(v) = d'$ is defined **do**
    for each u where $a(u)$ is undefined and $C_{uv} \in C$ **do**
        $D_u := \{d \in D_u \mid (d, d') \in C_{uv}\}$
return γ

Definition 42 (Arc consistency)**.** For a constraint network γ and a partial assignment a, a pair $(u, v) \in V^2$ of variables is arc consistent if for every value in the domain of u, there exists a valid partner in the domain for v. For a variable $w \in dom(a)$, assume $D_w = \{a(w)\}$.

Remark 5. Enforcing arc consistency (removing variable domain values until γ is arc consistent):

function Revise(γ,u,v) **returns** modified γ

for each $d \in D_u$ **do**
    if there is no $d' \in D_v$ with $(d, d') \in C_{uv}$ **then**
        $D_u := D_u \setminus \{d\}$
return γ

Remark 6. Runtime, if k is maximal domain size: $\mathcal{O}(k^2)$, based on implementation where the test "$(d, d') \in C_{uv}$?" is constant time.

AC-1 Iterate repeatedly over every constraint and enforce arc consistency in both directions. Stop when no changes have been made in one iteration.

function AC-1(γ) **returns** modified γ

repeat
    changesMade := False
    for each constraint C_{uv} **do**
        Revise(γ, u, v)
        if D_u reduces **then** changesMade := True
        Revise(γ, v, u)
        if D_v reduces **then** changesMade := True
until changesMade := False
return γ

Remark 7. Runtime, if n variables, m constraints, k maximal domain size: $\mathcal{O}(mk^2nk)$: mk^2 for each inner loop, fixed point reached at the latest once all nk variable values have been removed.

AC-3 For every constraint C_{uv}, put (u, v) and (v, u) in M. Do the following until $M = \emptyset$: Remove one element (u, v) from M and enforce arc consistency from u to v. If D_u changed, add (w, u) to M for every constraint C_{uw} and $w \neq v$.

function AC-3(γ) **returns** modified γ

$M := \emptyset$
for each constraint $C_{uv} \in C$ **do**
    $M := M \cup \{(u, v), (v, u)\}$
while $M \neq \emptyset$ **do**
    remove any element (u, v) from M
    Revise(γ, u, v)
    if D_u has changed in the call to revise **then**
        for each constraint $C_{wu} \in C$ where $w \neq v$ **do**
            $M := M \cup \{(w, u)\}$
return γ

Remark 8. Let $\gamma = \langle V, D, C\rangle$ be a constraint network with m constraints, and maximal domain size k. Then AC-3(γ) runs in time $\mathcal{O}(mk^3)$.

Remark 9. To solve an acyclic constraint network, enforce arc consistency with AC-3 and run backtracking with inference on the arc consistent network. This will find a solution without having to backtrack.

Definition 43 (Acyclic Constraint Graph)**.** Let $\gamma = \langle V, D, C\rangle$ be a constraint network with n variables and maximal domain size k, whose constraint graph is acyclic. Then we can find a solution for γ, or prove γ to be inconsistent, in time $\mathcal{O}(nk^2)$.

function AcyclicCG(γ)

1: Obtain a directed tree from γ's constraint graph, picking an arbitrary variable v as the root, and directing arcs outwards.
2: Order the variables topologically, i.e., such that each vertex is ordered before its children; denote that order by $v_1, \dots v_n$.
3: **for** $i := n, n-1, \dots, 2$ **do**
4: Revise($\gamma, v_{\text{parent}(i)}, v_i$)
5: **if** $D_{v_{\text{parent}(i)}} = \emptyset$ **then return** "inconsistent"
6: /* Now, every variable is arc consistent relative to its children */
7: Run BacktrackingWithInference with forward checking, using the variable order $v_1, \dots v_n$.

Definition 44 (Cutset conditioning)**.** Let $\gamma = \langle V, D, C \rangle$ be a constraint network, and $V_0 \subseteq V$. V_0 is a cutset for γ if the sub-graph of γ's constraint graph induced by $V \setminus V_0$ is acyclic. V_0 is optimal if its size is minimal among all cutsets for γ.

function CutsetConditioning(γ, V_0, a) **returns** a solution, or "inconsistent"

1: $\gamma :=$ a copy of γ
2: $\gamma' :=$ ForwardChecking(γ', a)
3: **if** ex. $v \in V_0$ s.t. $a(v)$ is undefined **then**
4: select such v
5: **else**
6: $a' :=$ AcyclicCG(γ')
7: **if** $a' \neq$ "inconsistent" **then return** $a \cup a'$
8: **else return** "inconsistent"
9: **for** each $d \in$ copy of D'_v in some order **do**
10: $a' := a \cup \{v = d\}$; $D'_v := \{d\}$;
11: $a'' :=$ CutsetConditioning(γ', V_0, a')
12: **if** $a' \neq$ "inconsistent" **then return** a''
13: **return** "inconsistent"

4 Logic

Definition 45 (Syntax)**.** Rules to decide what are legal formulas.

Definition 46 (Semantics)**.** Rules to decide whether a formula A is true for a given assignment or interpretation ϕ. Write $\phi \models A$ is A is true under ϕ or $\phi \not\models A$ if not.

Definition 47 (Entailment)**.** B is entailed by A iff for every interpretation that makes A true, B is true as well (For every model ϕ, $\phi \models A \implies \phi \models B$). Write $A \models B$.

Definition 48 (Deduction)**.** Which statements can be derived from A using a set of inference rules $\mathcal{C}$? $A \vdash_{\mathcal{C}} B$ means B can be derived from A in the calculus $\mathcal{C}$.

Definition 49 (Soundness)**.** A calculus $\mathcal{C}$ is sound if for all formulas A, B it is true that $A \vdash_{\mathcal{C}} B \implies A \models B$.

Definition 50 (Complete)**.** A calculus $\mathcal{C}$ is complete if for all formulas A, B it is true that $A \models B \implies A \vdash_{\mathcal{C}} B$.

Definition 51 (Propositional logic)**.** Also called PL^0. Let $\text{wff}_o(\mathcal{V}_o)$ be the set of well-formed formulas with variables $\mathcal{V}_o$.

Definition 52 (First order logic)**.** Also called *FOL*. $\text{wff}_\iota(\Sigma_\iota)$ is the set of well-formed terms over a signature Σ_ι (function and skolem constants). $\text{wff}_o(\Sigma)$ is the set of well-formed propositions over a signature Σ (Σ_ι plus connectives and predicate constants).

Theorem 2 (Unsatisfiability)**.** Let $\mathcal{H}$ be a set of formulas and A be a single formula. $\mathcal{H} \models A$ iff $\mathcal{H} \cup \{\neg A\}$ is unsatisfiable.

Definition 53 (Conjunctive normal form)**.** A formula A is in conjunctive normal form (CNF) if it is a conjunction of disjunctions of literals: $A = \wedge_{i=1}^{n} \vee_{j=1}^{m_i} l_{ij}$ where l_{ij} is a literal, which is a possibly negated variable.

Definition 54 (Natural deduction)**.** Also $\mathcal{ND}^1$. Contains the following inference rules:

$$\frac{\text{A} \qquad \text{B}}{A \wedge B} \wedge I \qquad \frac{\forall X.A}{[B/X](A)} \forall E$$

$$\frac{A \wedge B}{A} \wedge E_l \qquad \frac{[B/X](A)}{\exists X.A} \exists I$$

$$\frac{A \wedge B}{B} \wedge E_r$$

$$\frac{}{A \vee \neg A} \text{TND} \qquad \frac{\exists X.A \qquad \dfrac{[c/X](A)}{\text{B}}}{B} \exists E$$

$$\frac{\dfrac{\text{A}}{\text{B}}}{A \implies B} \implies I \qquad \frac{}{A = A} = I$$

$$\frac{A \implies B \qquad \text{A}}{\text{B}} \implies E$$

$$\frac{\text{A}}{\forall X.A} \forall I \qquad \frac{A = B \qquad C[A]_p}{[B/p](C)} = E$$

Definition 55 (Analytical tableaux)**.** Contains the following inference rules:

$$\frac{A \wedge B^T}{A^T, B^T} \mathcal{T}_0\wedge \qquad \frac{A \implies B^T}{A^F | B^T}$$

$$\frac{A \wedge B^F}{A^F | B^F} \mathcal{T}_0\vee \qquad \frac{A \implies B^F}{A^T, B^F}$$

$$\frac{\neg A^T}{A^F} \mathcal{T}_0^T\neg \qquad \frac{A^T \qquad A \implies B^T}{B^T}$$

$$\frac{\neg A^F}{A^T} \mathcal{T}_0^F\neg \qquad \frac{A \vee B^T}{A^T | B^T}$$

$$\frac{\neg A^T \qquad \neg A^F}{\bot} \mathcal{T}_0 cut \qquad \frac{A \vee B^F}{A^F, B^F}$$

$$\frac{\forall X.A^T \qquad C \in \text{cwff}_\iota(\Sigma_\iota)}{[C/X](A)^T} \mathcal{T}_1\forall \qquad \frac{A \Leftrightarrow B^T}{A^T, B^T | A^F, B^F}$$

$$\frac{\forall X.A^F \qquad c \in (\Sigma_0^{sk} \setminus \mathcal{H})}{[c/X](A)^F} \mathcal{T}_1\exists \qquad \frac{A \Leftrightarrow B^F}{A^T, B^F | A^F, B^T}$$

All rules do the same thing: If the premise is P, open one branch for every way to make P true. For example:

There are two ways to make the premise $A \implies B^T$ true. Either A^F or B^T. This means two branches are opened, one for A^F and one for B^T. (N.b. there are actually three ways to make the premise true: $A^T, B^T | A^F, B^T | A^F, B^F$. It is still enough to open just two branches, because B^T captures the former two ways and A^F captures the latter two.)

Of the first five rules, all operate in the same branch, except $\mathcal{T}_0\vee$. This rule creates two new branches, one for A^F and one for B^F.

A tableaux is saturated if all rules that can be applied do not contribute new material.

A branch is closed if it contains $\bot$, else open. A tableau is closed if all of it's branches are closed (n.b. a closed tableau may or may not be saturated).

Theorem 3. A is valid iff there is a closed tableau with A^F at the root.

Definition 56 (Resolution)**.** The resolution calculus has three rules of inference:

$$\frac{P^T \vee A \qquad P^F \vee B}{A \vee B}$$

$$\frac{P^T \vee A \qquad Q^F \vee B \qquad \sigma = mgu(P,Q)}{\sigma(A) \vee \sigma(B)}$$

$$\frac{P^\alpha \vee Q^\alpha \vee A \qquad \sigma = mgu(P,Q)}{\sigma(P) \vee \sigma(A)}$$

The last two rules are only necessary when working in FOL. The second rule assumes A, B share no variables. In the last rule, α is either $\top$ or $\bot$.

Remark 10 (Resolution)**.** A derivation of the empty clause $\square$ from a clause set S is a resolution refutation. This means S is unsatisfiable.

A resolution refutation of $CNF(\mathcal{H} \wedge \neg A)$ is a resolution proof for $\mathcal{H} \models A$. Refer to the unsatisfiability theorem.

Definition 57 (Conversion into CNF)**.** An arbitrary formula can be transformed into a CNF by doing the following steps:

1. Rewrite all implications and equivalences using negations, disjunctions and conjunction.
2. Move negations inwards. After this step, negations may only occur immediately before predicates.
3. Rename variables bound by quantifiers to make them unique. After this step, a variable is either free or bound by exactly one quantifier.
4. Move quantifiers outwards. After this step, a quantifier may only occur in the prefix of the formula but not inside of a $\wedge, \vee$ or $\neg$.
5. Replace variables bound by existential quantifiers with skolem functions. Example: $\forall x_1 \ldots \forall x_n \exists x_{n+1}.P$ is replaced by $\forall x_1 \ldots \forall x_n.[f(x_1, \ldots, x_n)/x_{n+1}](P)$.
6. Drop universal quantifiers.
7. Distribute $\vee$'s inwards over $\wedge$'s: $A \vee (B \wedge C)$ becomes $(A \vee B) \wedge (A \vee C)$.

Remark 11. $CNF(P)$ is satisfiable iff P is satisfiable.

Definition 58 (DPLL)**.** The DPLL procedure is an algorithm to find an interpretation satisfying a clause set Δ.

Definition 59 (Abstract consistency)**.** A family ∇ of sets of propositions is an abstract consistency class, iff for every $\phi \in \nabla$ it is true that

- $P \notin \phi$ or $\neg P \notin \phi$ for every variable P
- $\phi * A \in \nabla$ if $\neg\neg A \in \phi$
- $\phi * A \in \nabla$ or $\phi * B \in \nabla$ if $(A \vee B) \in \phi$
- $\phi * \neg A * \neg B \in \nabla$ if $\neg(A \vee B) \in \phi$
- $\phi * [B/X](A) \in \nabla$ for every closed term B if $(\forall X.A) \in \phi$
- $\phi * [c/X](A) \in \nabla$ for a fresh constant c if $(\neg\forall X.A) \in \phi$

Theorem 4 (Model existence)**.** If ∇ is an abstract consistency class and $\phi \in \nabla$ then ϕ is satisfiable.

Theorem 5. $\nabla = \{\phi \mid \phi \text{ has no closed tableau}\}$ is an abstract consistency class.

Theorem 6. The calculus of analytical tableaux is complete.

Proof. Show that whenever A is valid, $\neg A$ must have a closed tableau: Assume A is valid. Assume $\neg A$ has no closed tableau. $\neg A \in \nabla$. As per model existence, $\neg A$ is satisfiable because it is in ∇. This contradicts the assumption that A is valid. $\neg A$ must have a closed tableau. $\square$

5 Logic Programming

Definition 60 (Fact)**.** This is a term that is unconditionally true.

Definition 61 (Rule)**.** This is a term that is true if certain premises are true.

Definition 62 (Clause)**.** Facts and rules are both clauses.

Definition 63 (Knowledge base)**.** The knowledge base given by a Prolog program is the set of terms that can be derived from it using the following rules:

$$\frac{A \qquad A \implies B}{B}\ \text{MP} \qquad \frac{A \qquad B}{A \wedge B}\ \wedge I$$

$$\frac{A}{[B/X](A)}\ \text{Subst}$$

Definition 64 (Horn clause)**.** A horn clause is a clause with at most one positive literal.

Remark 12. The Prolog rule $H :- B_1, \ldots, B_n$ is the implication $B_1 \wedge \cdots \wedge B_n \implies H$ can be written as $\neg B_1 \vee \cdots \vee \neg B_n \vee H$. This is a horn clause.

6 Planning

Definition 65 (Planning language)**.** A planning language describes, using logic, the

- set of possible states
- initial state I
- goal condition G
- set of actions A in terms of preconditions and effects.

Remark 13. By describing the components of a planning problem using logic, the solver is able to gain insight into the problem structure. This means the solver is no longer working on a black-box.

Definition 66 (Satisficing planning)**.** A procedure that takes as input a planning problem Π and outputs a plan for Π or unsolvable, if no such plan exists.

Definition 67 (Optimal planning)**.** A procedure that takes as input a planning problem Π and outputs an optimal plan for Π or unsolvable, if no such plan exists.

Definition 68 (STRIPS planning task)**.** This is a encoding of a planning problem using a 4-tuple $\Pi = \langle P, A, I, G\rangle$ where

- P is a finite set of facts
- A is a finite set of actions, each given as a triple $\langle pre_a, add_a, del_a\rangle$. The components of the triple are called preconditions, add-list and delete-list
- $I \subseteq P$ is the initial state
- $G \subseteq P$ is the goal.

Definition 69 (Only-adds relaxation)**.** This is a relaxation of a given STRIPS planning task Π where P, I, G are the same and the actions are the actions of Π with empty preconditions and empty delete lists.

Definition 70 (Delete relaxation)**.** This is a relaxation Π^+ of a given STRIPS planning task Π where P, I, G are the same and the actions are the actions of Π with empty delete lists.

Definition 71 (Relaxed plan)**.** Let $\Pi = \langle P, A, I, G\rangle$ be a planning task. A relaxed plan for $s \in \mathcal{P}(P)$ is a plan for $\langle P, A, s, G\rangle^+$. A relaxed plan for I is called a relaxed plan for Π.

Remark 14 (Planning algorithms)**.** Plans for planning tasks can be found using informed search strategies.

Definition 72 (h^+-heuristic)**.** For a planning task $\Pi = \langle P, A, I, G\rangle$, $h^+ : \mathcal{P} \mapsto \mathbb{N} \cup \{\infty\}$ is a heuristic calculating the length of the optimal relaxed plan for s or ∞ if no plan exists.

Theorem 7. h^+ is admissible.

Theorem 8. Calculating h^+ is NP-complete.

Definition 73 (h^{FF}-heuristic)**.** For a planning task $\Pi = \langle P, A, I, G\rangle$, $h^{FF} : \mathcal{P} \mapsto \mathbb{N} \cup \{\infty\}$ is a heuristic calculating the length of a relaxed plan for s or ∞ if no plan exists.

Remark 15. h^{FF} never underestimates h^+ and may even overestimate h^*. Thus, h^{FF} is not admissible and may not be used for optimal planning (but for satisficing planning).