DEDUCTION SYSTEMS

Optimizations for Tableau Procedures

Sebastian Rudolph
Agenda

- Recap Tableau Calculus
- Optimizations
  - Unfolding
  - Absorption
  - Dependency-Directed Backtracking
  - Further Optimizations
- Classification
- Summary
Agenda

- Recap Tableau Calculus
- Optimizations
  - Unfolding
  - Absorption
  - Dependency-Directed Backtracking
  - Further Optimizations
- Classification
- Summary
Tableau Algorithm for $\mathcal{ALC}$ Concepts and TBoxes

- check satisfiability of $C$ by constructing an abstraction of a model $\mathcal{I}$ such that $C^\mathcal{I} \neq \emptyset$
Tableau Algorithm for $\mathcal{ALC}$ Concepts and TBoxes

- check satisfiability of $C$ by constructing an abstraction of a model $\mathcal{I}$ such that $C^\mathcal{I} \neq \emptyset$
- concepts in negation normal form (NNF) $\iff$ makes rules simpler
Tableau Algorithm for \( \mathcal{ALC} \) Concepts and TBoxes

- check satisfiability of \( C \) by constructing an abstraction of a model \( \mathcal{I} \) such that \( C^\mathcal{I} \neq \emptyset \)
- concepts in negation normal form (NNF) \( \leadsto \) makes rules simpler
- tableau (model abstraction) corresponds to a graph/tree \( G = \langle V, E, L \rangle \)
Tableau Algorithm for $\mathcal{ALC}$ Concepts and TBoxes

- check satisfiability of $C$ by constructing an abstraction of a model $\mathcal{I}$ such that $C^\mathcal{I} \neq \emptyset$
- concepts in negation normal form (NNF) $\Rightarrow$ makes rules simpler
- tableau (model abstraction) corresponds to a graph/tree $G = \langle V, E, L \rangle$
- initialize $G$ with a node $v$ such that $L(v) = \{C\}$
Tableau Algorithm for $\mathcal{ALC}$ Concepts and TBoxes

- check satisfiability of $C$ by constructing an abstraction of a model $\mathcal{I}$ such that $C^\mathcal{I} \neq \emptyset$
- concepts in negation normal form (NNF) $\Leftrightarrow$ makes rules simpler
- tableau (model abstraction) corresponds to a graph/tree $G = \langle V, E, L \rangle$
- initialize $G$ with a node $v$ such that $L(v) = \{C\}$
- extend $G$ by applying tableau rules
Tableau Algorithm for $\mathcal{ALC}$ Concepts and TBoxes

- check satisfiability of $C$ by constructing an abstraction of a model $\mathcal{I}$ such that $C^\mathcal{I} \neq \emptyset$
- concepts in negation normal form (NNF) $\Rightarrow$ makes rules simpler
- tableau (model abstraction) corresponds to a graph/tree $G = \langle V, E, L \rangle$
- initialize $G$ with a node $v$ such that $L(v) = \{C\}$
- extend $G$ by applying tableau rules
  - $\Box$-rule non-deterministic (we guess)
- tableau branch closed if $G$ contains an atomic contradiction (clash)
Tableau Algorithm for $\mathcal{ALC}$ Concepts and TBoxes

- check satisfiability of $C$ by constructing an abstraction of a model $\mathcal{I}$ such that $C^\mathcal{I} \neq \emptyset$
- concepts in negation normal form (NNF) $\leadsto$ makes rules simpler
- tableau (model abstraction) corresponds to a graph/tree $G = \langle V, E, L \rangle$
- initialize $G$ with a node $v$ such that $L(v) = \{C\}$
- extend $G$ by applying tableau rules
  - $\sqcup$-rule non-deterministic (we guess)
- tableau branch closed if $G$ contains an atomic contradiction (clash)
- tableau construction successful, if no further rules are applicable and there is no contradiction
Tableau Algorithm for $\mathcal{ALC}$ Concepts and TBoxes

- check satisfiability of $C$ by constructing an abstraction of a model $\mathcal{I}$ such that $C^\mathcal{I} \neq \emptyset$
- concepts in negation normal form (NNF) $\Rightarrow$ makes rules simpler
- tableau (model abstraction) corresponds to a graph/tree $G = \langle V, E, L \rangle$
- initialize $G$ with a node $v$ such that $L(v) = \{C\}$
- extend $G$ by applying tableau rules
  - $\sqcup$-rule non-deterministic (we guess)
- tableau branch closed if $G$ contains an atomic contradiction (clash)
- tableau construction successful, if no further rules are applicable and there is no contradiction
- $C$ is satisfiable iff there is a successful tableau construction
we condense the TBox into one concept:
for \( T = \{ C_i \sqsubseteq D_i \mid 1 \leq i \leq n \} \), \( C_T = \text{NNF}(\bigcap_{1 \leq i \leq n} \neg C_i \cup D_i) \)

we extend the rules of the \( \mathcal{ALC} \) tableau algorithm:

\( T \)-rule: for an arbitrary \( v \in V \) with \( C_T \notin L(v) \),
let \( L(v) := L(v) \cup \{ C_T \} \).

in order to take an ABox \( A \) into account, initialize \( G \) such that

- \( V \) contains a node \( v_a \) for every individual \( a \) in \( A \)
- \( L(v_a) = \{ C \mid C(a) \in A \} \)
- \( \langle v_a, v_b \rangle \in E \) iff \( r(a, b) \in A \)
Extensions of the Logic

- plus inverses ($ALCI$): inverse roles in edge labels, definition and use of $r$-neighbors instead of $r$-successors in tableau rules
- plus functional roles ($ALCIF$): merging of nodes to account for functionality

blocking guarantees termination:
- $ALC$ subset-blocking
- plus inverses ($ALCI$): equality blocking
- plus functional roles ($ALCIF$): pairwise blocking
Agenda

- Recap Tableau Calculus
- **Optimizations**
  - Unfolding
  - Absorption
  - Dependency-Directed Backtracking
  - Further Optimizations
- Classification
- Summary
Optimizations

- Naïve implementation not performant enough
  - $\mathcal{T}$-regel adds one disjunction per axiom to the corresponding node
  - ontologies may contain $> 1.000$ axioms and tableaux may contain thousands of nodes
Optimizations

- Naïve implementation not performant enough
  - $T$-regel adds one disjunction per axiom to the corresponding node
  - ontologies may contain $> 1,000$ axioms and tableaux may contain thousands of nodes

- realistic implementations use many optimizations
  - (Lazy) unfolding
  - Absorbtion
  - Dependency directed backtracking
  - Simplification and Normalization
  - Caching
  - Heuristics
  - ...
Optimizations

- Naïve implementation not performant enough
  - $\mathcal{T}$-regel adds one disjunction per axiom to the corresponding node
  - ontologies may contain $>1,000$ axioms and tableaux may contain thousands of nodes

- realistic implementations use many optimizations
  - (Lazy) unfolding
  - Absorbtion
  - Dependency directed backtracking
  - Simplification and Normalization
  - Caching
  - Heuristics
  - ...
Agenda

• Recap Tableau Calculus

• Optimizations
  – Unfolding
  – Absorption
  – Dependency-Directed Backtracking
  – Further Optimizations

• Classification

• Summary
Unfolding

- $\mathcal{T}$-rule is not necessary if $\mathcal{T}$ is unfoldable, i.e., every axiom is:
  - definitorial: form $A \sqsubseteq C$ or $A \equiv C$ for $A$ a concept name
    ($A \equiv C$ corresponds to $A \sqsubseteq C$ and $C \sqsubseteq A$)
  - acyclic: $C$ uses $A$ neither directly nor indirectly
  - unique: only one such axiom exists for every concept name $A$
Unfolding

- $\mathcal{T}$-rule is not necessary if $\mathcal{T}$ is unfoldable, i.e., every axiom is:
  - definitorial: form $A \sqsubseteq C$ or $A \equiv C$ for $A$ a concept name ($A \equiv C$ corresponds to $A \sqsubseteq C$ and $C \sqsubseteq A$)
  - acyclic: $C$ uses $A$ neither directly nor indirectly
  - unique: only one such axiom exists for every concept name $A$
- If $\mathcal{T}$ is unfoldable, the TBox can be (unfolded) into a concept
Unfolding Example

- We check satisfiability of $A$ w.r.t. the TBox $\mathcal{T}$

\[
\mathcal{T}:
\begin{align*}
A & \sqsubseteq B \sqcap \exists r.C \\
B & \equiv C \sqcup D \\
C & \sqsubseteq \exists r.D
\end{align*}
\]
Unfolding Example

- We check satisfiability of $A$ w.r.t. the TBox $\mathcal{T}$

\[
\mathcal{T}:
\]

\[
A \sqsubseteq B \sqcap \exists r.C
\]

\[
B \equiv C \sqcup D
\]

\[
C \sqsubseteq \exists r.D
\]
Unfolding Example

- We check satisfiability of $A$ w.r.t. the TBox $\mathcal{T}$

$$\mathcal{T}:$$
- $A \sqsubseteq B \sqcap \exists r.C$
- $B \equiv C \sqcup D$
- $C \sqsubseteq \exists r.D$

$$A \quad \sim A \sqcap B \sqcap \exists r.C$$
Unfolding Example

• We check satisfiability of $A$ w.r.t. the TBox $\mathcal{T}$

\[
\begin{align*}
\mathcal{T}: & \\
A & \sqsubseteq B \sqcap \exists r.C \\
B & \equiv C \sqcup D \\
C & \sqsubseteq \exists r.D \\
\end{align*}
\]

\[
\begin{align*}
A & \\
\neg A \sqcap B \sqcap \exists r.C \\
\neg A \sqcap (C \sqcup D) \sqcap \exists r.C \\
\end{align*}
\]
Unfolding Example

- We check satisfiability of $A$ w.r.t. the TBox $\mathcal{T}$

\[
\begin{align*}
A & \\
\sim A \cap B \cap \exists r. C & \\
\sim A \cap (C \cup D) \cap \exists r. C & \\
\sim A \cap ((C \cap \exists r. D) \cup D) \cap \exists r. (C \cap \exists r. D) & \\
\end{align*}
\]

$\mathcal{T}$:

\[
\begin{align*}
A & \sqsubseteq B \sqcap \exists r. C & \\
B & \equiv C \sqcup D & \\
C & \sqsubseteq \exists r. D & \\
\end{align*}
\]
Unfolding Example

- We check satisfiability of $A$ w.r.t. the TBox $\mathcal{T}$

  $\mathcal{T}$:
  
  $A \sqsubseteq B \sqcap \exists r. C$
  
  $B \equiv C \sqcup D$
  
  $C \sqsubseteq \exists r. D$

- $A$ is satisfiable w.r.t. $\mathcal{T}$ iff

  $A \sqcap ((C \sqcap \exists r. D) \sqcup D) \sqcap \exists r. (C \sqcap \exists r. D)$

  is satisfiable w.r.t. the empty TBox
We obtain the following contradiction-free tableau for the satisfiability of
\[ U = A \cap \((C \cap \exists r.D) \cup D) \cap \exists r.(C \cap \exists r.D) : \]

\[
L(v_0) = \{U, A, (C \cap \exists r.D) \cup D, \exists r.(C \cap \exists r.D), C \cap \exists r.D, C, \exists r.D\}
\]
\[
L(v_1) = \{C \cap \exists r.D, C, \exists r.D\}
\]
\[
L(v_2) = \{D\}
\]
\[
L(v_3) = \{D\}
\]
We obtain the following contradiction-free tableau for the satisfiability of
\[ U = A \cap ((C \cap \exists r.D) \cup D) \cap \exists r.(C \cap \exists r.D) : \]

\[
\begin{align*}
L(v_0) & = \{ U, A, (C \cap \exists r.D) \cup D, \\
& \quad \exists r.(C \cap \exists r.D), C \cap \exists r.D, \\
& \quad C, \exists r.D \} \\
L(v_1) & = \{ C \cap \exists r.D, C, \exists r.D \} \\
L(v_2) & = \{ D \} \\
L(v_3) & = \{ D \}
\end{align*}
\]

Only one disjunctive decision left!
Lazy Unfolding

- computation of NNF together with unfolding may decrease performance, e.g.:
  - satisfiability of $C \sqcap \neg C$ w.r.t. $\mathcal{T} = \{C \sqsubseteq A \sqcap B\}$
  - unfolding: $C \sqcap A \sqcap B \sqcap \neg (C \sqcap A \sqcap B)$
  - NNF + unfolding: $C \sqcap A \sqcap B \sqcap (\neg C \sqcup \neg A \sqcup \neg B)$
Lazy Unfolding

- computation of NNF together with unfolding may decrease performance, e.g.:
  - satisfiability of \( C \cap \neg C \) w.r.t. \( \mathcal{T} = \{ C \sqsubseteq A \sqcap B \} \)
  - unfolding: \( C \cap A \cap B \cap \neg (C \cap A \cap B) \)
  - NNF + unfolding: \( C \cap A \cap B \cap (\neg C \sqcup \neg A \sqcup \neg B) \)

- better: apply NNF and unfolding if needed, via corresponding tableau rules:
  - \( A \equiv C \leadsto A \sqsubseteq C \) and \( A \sqsupseteq C \)

\( \sqsubseteq \)-rule: For \( v \in V \) such that \( A \sqsubseteq C \in \mathcal{T} \), \( A \in L(v) \) and \( C \notin L(v) \)
  let \( L(v) := L(v) \cup C \).

\( \sqsupseteq \)-rule: For \( v \in V \) such that \( A \sqsupseteq C \in \mathcal{T} \), \( \neg A \in L(v) \) and \( \neg C \notin L(v) \)
  let \( L(v) := L(v) \cup \{ \neg C \} \).

\( \neg \)-rule: For \( v \in V \) such that \( \neg C \in L(v) \) and \( \text{NNF}(\neg C) \notin L(v) \),
  let \( L(v) := L(v) \cup \{ \text{NNF}(\neg C) \} \).
Agenda

• Recap Tableau Calculus
• Optimizations
  – Unfolding
  – Absorption
  – Dependency-Directed Backtracking
  – Further Optimizations

• Classification

• Summary
Absorption

- What if $\mathcal{T}$ is not unfoldable?
  - Separate $\mathcal{T}$ into $\mathcal{T}_u$ (unfoldable part) and $\mathcal{T}_g$ (GCI, not unfoldable)
  - $\mathcal{T}_u$ is treated via $\sqsubseteq$- and $\sqsupseteq$-rules
  - $\mathcal{T}_g$ is treated via the $\mathcal{T}$-rule
Absorption

• What if $\mathcal{T}$ is not unfoldable?
  – Separate $\mathcal{T}$ into $\mathcal{T}_u$ (unfoldable part) and $\mathcal{T}_g$ (GCl, not unfoldable)
  – $\mathcal{T}_u$ is treated via $\sqsubseteq$- and $\sqsupseteq$-rules
  – $\mathcal{T}_g$ is treated via the $\mathcal{T}$-rule

• absorption decreases $\mathcal{T}_g$ and increases $\mathcal{T}_u$
  1. take an axiom from $\mathcal{T}_g$, e.g., $A \sqcap B \sqsubseteq C$
  2. transform the axiom: $A \sqsubseteq C \sqcup \neg B$
  3. if $\mathcal{T}_u$ contains an axiom of the form $A \equiv D$ ($A \sqsubseteq D$ and $D \sqsupseteq A$),
    then $A \sqsubseteq C \sqcup \neg B$ cannot be absorbed;
    $A \sqsubseteq C \sqcup \neg B$ remains in $\mathcal{T}_g$
  4. otherwise, if $\mathcal{T}_u$ contains an axiom of the form $A \sqsubseteq D$,
    then absorb $A \sqsubseteq C \sqcup \neg B$ resulting in $A \sqsubseteq D \sqcap (C \sqcup \neg B)$
  5. otherwise move $A \sqsubseteq C \sqcup \neg B$ to $\mathcal{T}_u$
Absorption

- What if $\mathcal{T}$ is not unfoldable?
  - Separate $\mathcal{T}$ into $\mathcal{T}_u$ (unfoldable part) and $\mathcal{T}_g$ (GCIs, not unfoldable)
  - $\mathcal{T}_u$ is treated via $\sqsubseteq$- and $\sqsupseteq$-rules
  - $\mathcal{T}_g$ is treated via the $\mathcal{T}$-rule

- absorption decreases $\mathcal{T}_g$ and increases $\mathcal{T}_u$
  1. take an axiom from $\mathcal{T}_g$, e.g., $A \sqcap B \subseteq C$
  2. transform the axiom: $A \sqsubseteq C \sqcup \neg B$
  3. if $\mathcal{T}_u$ contains an axiom of the form $A \equiv D$ ($A \sqsubseteq D$ and $D \sqsupseteq A$), then $A \sqsubseteq C \sqcup \neg B$ cannot be absorbed; $A \sqsubseteq C \sqcup \neg B$ remains in $\mathcal{T}_g$
  4. otherwise, if $\mathcal{T}_u$ contains an axiom of the form $A \sqsubseteq D$, then absorb $A \sqsubseteq C \sqcup \neg B$ resulting in $A \sqsubseteq D \sqcap (C \sqcup \neg B)$
  5. otherwise move $A \sqsubseteq C \sqcup \neg B$ to $\mathcal{T}_u$

- If $A \equiv D \in \mathcal{T}_u$, try rewriting/absorption with other axioms in $\mathcal{T}_u$
Absorption

- What if $T$ is not unfoldable?
  - Separate $T$ into $T_u$ (unfoldable part) and $T_g$ (GCIs, not unfoldable)
  - $T_u$ is treated via $\sqsubseteq$- and $\sqsupseteq$-rules
  - $T_g$ is treated via the $T$-rule
- absorption decreases $T_g$ and increases $T_u$
  1. take an axiom from $T_g$, e.g., $A \cap B \sqsubseteq C$
  2. transform the axiom: $A \sqsubseteq C \sqcup \neg B$
  3. if $T_u$ contains an axiom of the form $A \equiv D$ ($A \sqsubseteq D$ and $D \sqsupseteq A$),
     then $A \sqsubseteq C \sqcup \neg B$ cannot be absorbed;
     $A \sqsubseteq C \sqcup \neg B$ remains in $T_g$
  4. otherwise, if $T_u$ contains an axiom of the form $A \sqsubseteq D$,
     then absorb $A \sqsubseteq C \sqcup \neg B$ resulting in $A \sqsubseteq D \sqcap (C \sqcup \neg B)$
  5. otherwise move $A \sqsubseteq C \sqcup \neg B$ to $T_u$
- If $A \equiv D \in T_u$, try rewriting/absorption with other axioms in $T_u$
- nondeterministic: $B \sqsubseteq C \sqcup \neg A$ also possible
Agenda

• Recap Tableau Calculus
• Optimizations
  – Unfolding
  – Absorption
  – Dependency-Directed Backtracking
  – Further Optimizations
• Classification
• Summary
Dependency-Directed Backtracking

- despite those optimizations, search space often too big
- let $v \in V$ with $(C_1 \sqcup D_1) \cap \ldots \cap (C_n \sqcup D_n) \cap \exists r \cdot \neg A \cap \forall r. A \in L(v)$
Dependency-Directed Backtracking

- despite those optimizations, search space often too big
- let $v \in V$ with $(C_1 \sqcup D_1) \sqcap \ldots \sqcap (C_n \sqcup D_n) \sqcap \exists r. \neg A \sqcap \forall r. A \in L(v)$

$v$-rule $L(v) := L(v) \cup \{(C_1 \sqcup D_1), \ldots, (C_n \sqcup D_n), \exists r. \neg A, \forall r. A\}$
Dependency-Directed Backtracking

- despite those optimizations, search space often too big
- let \( v \in V \) with \( (C_1 \sqcup D_1) \cap \ldots \cap (C_n \sqcup D_n) \cap \exists r. \lnot A \cap \forall r. A \in L(v) \)

\[
\begin{align*}
\sqcap \text{-rule } & \quad L(v) := L(v) \cup \{(C_1 \sqcup D_1), \ldots, (C_n \sqcup D_n), \exists r. \lnot A, \forall r. A\} \\
\sqcup \text{-rule } & \quad L(v) := L(v) \cup \{C_1\} \\
\vdots & \quad \vdots \quad \vdots \\
\sqcup \text{-rule } & \quad L(v) := L(v) \cup \{C_n\}
\end{align*}
\]
Dependency-Directed Backtracking

- despite those optimizations, search space often too big
- let \( v \in V \) with \((C_1 \sqcup D_1) \cap \ldots \cap (C_n \sqcup D_n) \cap \exists r. \neg A \cap \forall r.A \in L(v)\)

\[
\begin{align*}
\cap -rule \quad L(v) & := L(v) \cup \{(C_1 \sqcup D_1), \ldots, (C_n \sqcup D_n), \exists r. \neg A, \forall r.A\} \\
\cup -rule \quad L(v) & := L(v) \cup \{C_1\} \\
\vdots \quad \vdots \quad \vdots \\
\cup -rule \quad L(v) & := L(v) \cup \{C_n\} \\
\exists -rule \quad L(w) & := \{\neg A\}
\end{align*}
\]
Dependency-Directed Backtracking

• despite those optimizations, search space often to big
• let $v \in V$ with $(C_1 \sqcup D_1) \cap \ldots \cap (C_n \sqcup D_n) \cap \exists r. \neg A \cap \forall r. A \in L(v)$

\[ \sqcap \text{-rule} \quad L(v) := L(v) \cup \{(C_1 \sqcup D_1), \ldots, (C_n \sqcup D_n), \exists r. \neg A, \forall r. A\} \]

\[ \sqcup \text{-rule} \quad L(v) := L(v) \cup \{C_1\} \]

\[ \vdots \quad \vdots \quad \vdots \]

\[ \sqcup \text{-rule} \quad L(v) := L(v) \cup \{C_n\} \]

\[ \exists \text{-rule} \quad L(w) := \{\neg A\} \]

\[ \forall \text{-rule} \quad L(w) := \{-A, A\} \quad \text{clash} \]
Dependency-Directed Backtracking

- despite those optimizations, search space often too big
- let \( v \in V \) with \((C_1 \sqcup D_1) \sqcap \ldots \sqcap (C_n \sqcup D_n) \sqcap \exists r. \neg A \sqcap \forall r. A \in L(v)\)

\[
\begin{align*}
\sqcap \text{-rule} & \quad L(v) := L(v) \cup \{(C_1 \sqcup D_1), \ldots, (C_n \sqcup D_n), \\
& \quad \exists r. \neg A, \forall r. A\}
\\
\sqcup \text{-rule} & \quad L(v) := L(v) \cup \{C_1\}
\\
\vdots & \quad \vdots & \quad \vdots
\\
\sqcup \text{-rule} & \quad L(v) := L(v) \cup \{C_n\}
\\
\exists \text{-rule} & \quad L(v) := \{\neg A\}
\\
\forall \text{-rule} & \quad L(v) := \{\neg A, A\} \text{ clash}
\end{align*}
\]
Dependency-Directed Backtracking

- despite those optimizations, search space often to big
- let \( v \in V \) with \( (C_1 \sqcup D_1) \cap \ldots \cap (C_n \sqcup D_n) \cap \exists r. \neg A \cap \forall r. A \in L(v) \)

\[
\begin{align*}
\sqcap \text{-rule } L(v) & := L(v) \cup \{(C_1 \sqcup D_1), \ldots, (C_n \sqcup D_n), \\
& \quad \exists r. \neg A, \forall r. A\} \\
\sqcup \text{-rule } L(v) & := L(v) \cup \{C_1\} \\
& \vdots \\
\sqcup \text{-rule } L(v) & := L(v) \cup \{C_n\} \\
\exists \text{-rule } L(v) & := \left\{\neg A\right\} \\
\forall \text{-rule } L(v) & := \left\{\neg A, A\right\} \text{ clash} \\
\sqcup \text{-rule } L(v) & := L(v) \cup \{D_n\}
\end{align*}
\]
Dependency-Directed Backtracking

- despite those optimizations, search space often too big
- let $v \in V$ with $(C_1 \sqcup D_1) \cap \ldots \cap (C_n \sqcup D_n) \cap \exists r. \neg A \cap \forall r. A \in L(v)$

\[
\begin{align*}
\sqcap\text{-rule} & \quad L(v) := L(v) \cup \{(C_1 \sqcup D_1), \ldots, (C_n \sqcup D_n), \\
& \quad \exists r. \neg A, \forall r. A\}
\end{align*}
\]

\[
\begin{align*}
\sqcup\text{-rule} & \quad L(v) := L(v) \cup \{C_1\}
\end{align*}
\]

\[
\vdots \quad \vdots \quad \vdots
\]

\[
\begin{align*}
\sqcap\text{-rule} & \quad L(v) := L(v) \cup \{D_n\}
\end{align*}
\]

\[
\begin{align*}
\exists\text{-rule} & \quad L(v) := \\{\neg A\}
\end{align*}
\]

\[
\begin{align*}
\forall\text{-rule} & \quad L(v) := \\{\neg A, A\} \quad \text{clash}
\end{align*}
\]

\[
\begin{align*}
\sqcup\text{-rule} & \quad L(v) := L(v) \cup \{D_n\}
\end{align*}
\]

\[
\begin{align*}
\exists\text{-rule} & \quad L(w) := \\{\neg A\}
\end{align*}
\]
Dependency-Directed Backtracking

- despite those optimizations, search space often too big
- let $v \in V$ with $(C_1 \sqcup D_1) \cap \ldots \cap (C_n \sqcup D_n) \cap \exists r. \neg A \cap \forall r. A \in L(v)$

\[
\begin{align*}
\sqcap \text{-rule } & L(v) := L(v) \cup \{ (C_1 \sqcup D_1), \ldots, (C_n \sqcup D_n), \\
& \exists r. \neg A, \forall r. A \} \\
\sqcup \text{-rule } & L(v) := L(v) \cup \{ C_1 \} \\
\vdots & \vdots \vdots \\
\sqcup \text{-rule } & L(v) := L(v) \cup \{ C_n \} \\
\exists \text{-rule } & L(v) := \{ \neg A \} \\
\forall \text{-rule } & L(w) := \{ \neg A, A \} \text{ clash} \\
\exists \text{-rule } & L(w) := \{ \neg A \} \\
\forall \text{-rule } & L(w) := \{ \neg A, A \} \text{ clash}
\end{align*}
\]
Dependency-Directed Backtracking

- despite those optimizations, search space often too big
- let $v \in V$ with $(C_1 \sqcup D_1) \cap \ldots \cap (C_n \sqcup D_n) \cap \exists r. \neg A \cap \forall r. A \in L(v)$

$$\sqcap$$-rule
$$L(v) := L(v) \cup \{(C_1 \sqcup D_1), \ldots, (C_n \sqcup D_n), \exists r. \neg A, \forall r. A\}$$

$$\sqcup$$-rule
$$L(v) := L(v) \cup \{C_1\}$$

$$\exists$$-rule
$$L(w) := \{\neg A\}$$

$$\forall$$-rule
$$L(w) := \{\neg A, A\}$$ clash

$$\sqcap$$-rule
$$L(v) := L(v) \cup \{D_n\}$$

- exponentially big search space is traversed
Dependency-Directed Backtracking

- goal: recognize bad branching decisions quickly and do not repeat them
Dependency-Directed Backtracking

- goal: recognize bad branching decisions quickly and do not repeat them
- most frequently used: backjumping
Dependency-Directed Backtracking

- goal: recognize bad branching decisions quickly and do not repeat them
- most frequently used: backjumping
- backjumping works roughly as follows:
  - concepts in the node label are tagged with a set of integers (dependency set) allowing to identify the concept’s “origin”
  - initially, all concepts are tagged with $\emptyset$
  - tableau rules combine and extend these tags
  - $\squnion$-rule adds the tag $\{d\}$ to the existing tag, where $d$ is the $\sqcup$-depth (number of $\sqcup$-rules applied by now)
  - when encountering a contradiction, the labels allow to identify the origin of the concepts causing the contradiction
  - jump back to the last relevant application of a $\sqcup$-rule
Dependency-Directed Backtracking

- goal: recognize bad branching decisions quickly and do not repeat them
- most frequently used: backjumping
- backjumping works roughly as follows:
  - concepts in the node label are tagged with a set of integers (dependency set) allowing to identify the concept’s “origin”
  - initially, all concepts are tagged with $\emptyset$
  - tableau rules combine and extend these tags
  - $\sqcup$-rule adds the tag $\{d\}$ to the existing tag, where $d$ is the $\sqcup$-depth (number of $\sqcup$-rules applied by now)
  - when encountering a contradiction, the labels allow to identify the origin of the concepts causing the contradiction
  - jump back to the last relevant application of a $\sqcup$-rule
- irrelevant part of the search space is not considered
Dependency-Directed Backtracking

Example

\[(C_1 \sqcup D_1) \cap \ldots \cap (C_n \sqcup D_n) \cap \exists r. \neg A \cap \forall r. A \in L(v) \quad \text{tagged with } \emptyset\]
Dependency-Directed Backtracking

Example

\[(C_1 \sqcup D_1) \sqcap \ldots \sqcap (C_n \sqcup D_n) \sqcap \exists r. \neg A \sqcap \forall r. A \in L(v) \quad \text{tagged with } \emptyset\]

\[v \quad \sqcap \text{-rule} \quad L(v) \ := \ L(v) \cup \{(C_1 \sqcup D_1), \ldots, (C_n \sqcup D_n), \exists r. \neg A, \forall r. A\} \quad \text{all with } \emptyset\]
Dependency-Directed Backtracking

Example

\[(C_1 \sqcup D_1) \sqcap \ldots \sqcap (C_n \sqcup D_n) \sqcap \exists r. \neg A \sqcap \forall r. A \in L(v) \text{ tagged with } \emptyset\]

\[
\sqcap -\text{rule } L(v) := L(v) \cup \{(C_1 \sqcup D_1), \ldots, (C_n \sqcup D_n), \exists r. \neg A, \forall r. A\} \text{ all with } \emptyset
\]

\[
\sqcup -\text{rule } L(v) := L(v) \cup \{C_1\} \text{ } C_1 \text{ tagged with } \{1\}
\]

\[
\vdots \quad \vdots \quad \vdots
\]

\[
\sqcup -\text{rule } L(v) := L(v) \cup \{C_n\} \text{ } C_n \text{ tagged with } \{n\}
\]
Dependency-Directed Backtracking

Example

\((C_1 \sqcup D_1) \sqcap \ldots \sqcap (C_n \sqcup D_n) \sqcap \exists r. \neg A \sqcap \forall r. A \in L(v)\) tagged with \(\emptyset\)

\(\sqcap\text{-rule} \quad L(v) := L(v) \cup \{(C_1 \sqcup D_1), \ldots, (C_n \sqcup D_n), \exists r. \neg A, \forall r. A\}\) all with \(\emptyset\)

\(\sqcup\text{-rule} \quad L(v) := L(v) \cup \{C_1\} \quad C_1\text{ tagged with }\{1\}\)

\(\vdots \quad \vdots \quad \vdots\)

\(\sqcup\text{-rule} \quad L(v) := L(v) \cup \{C_n\} \quad C_n\text{ tagged with }\{n\}\)

\(\exists\text{-rule} \quad L(w) := \{\neg A\} \quad A, r\text{ tagged with }\emptyset\)
Dependency-Directed Backtracking

Example

\[(C_1 \sqcup D_1) \sqcap \ldots \sqcap (C_n \sqcup D_n) \sqcap \exists r. \neg A \sqcap \forall r. A \in L(v) \quad \text{tagged with } \emptyset\]

\(\sqcap\)-rule \quad L(v) := L(v) \cup \{(C_1 \sqcup D_1), \ldots, (C_n \sqcup D_n), \exists r. \neg A, \forall r. A\} \quad \text{all with } \emptyset

\(\sqcup\)-rule \quad L(v) := L(v) \cup \{C_1\} \quad C_1 \text{ tagged with } \{1\}

\(\sqcap\)-rule \quad L(v) := L(v) \cup \{C_n\} \quad C_n \text{ tagged with } \{n\}

\(\exists\)-rule \quad L(w) := \{\neg A\} \quad A, r \text{ tagged with } \emptyset

\(\forall\)-rule \quad L(w) := \{\neg A, A\} \quad \neg A \text{ tagged with mit } \emptyset
Dependency-Directed Backtracking

Example

\[(C_1 \sqcup D_1) \cap \ldots \cap (C_n \sqcup D_n) \cap \exists r. \neg A \cap \forall r. A \in L(v) \quad \text{tagged with } \emptyset\]

\[L(v) := L(v) \cup \{C_1 \sqcup D_1, \ldots, C_n \sqcup D_n\}, \exists r. \neg A, \forall r. A\] all with \(\emptyset\)

\[L(v) := L(v) \cup \{C_1\} \quad C_1 \text{ tagged with } \{1\}\]

\[L(v) := L(v) \cup \{C_n\} \quad C_n \text{ tagged with } \{n\}\]

\[\exists \text{-rule} \quad L(w) := \{\neg A\} \quad A, r \text{ tagged with } \emptyset\]

\[\forall \text{-rule} \quad L(w) := \{\neg A, A\} \quad \text{clash} \quad \neg A \text{ tagged with mit } \emptyset\]
Dependency-Directed Backtracking

Example

\[(C_1 \sqcup D_1) \cap \ldots \cap (C_n \sqcup D_n) \cap \exists r. \neg A \cap \forall r. A \in L(v) \quad \text{tagged with } \emptyset\]

\[
\begin{align*}
\sqcap \text{-rule} & \quad L(v) := L(v) \cup \{(C_1 \sqcup D_1), \ldots, (C_n \sqcup D_n), \\
& \quad \exists r. \neg A, \forall r. A\} \quad \text{all with } \emptyset \\
\sqcup \text{-rule} & \quad L(v) := L(v) \cup \{C_1\} \quad C_1 \text{ tagged with } \{1\}
\end{align*}
\]

\[
\begin{align*}
\sqcup \text{-rule} & \quad L(v) := L(v) \cup \{C_n\} \quad C_n \text{ tagged with } \{n\} \\
\exists \text{-rule} & \quad L(w) := \{\neg A\} \quad A, r \text{ tagged with } \emptyset \\
\forall \text{-rule} & \quad L(w) := \{\neg A, A\} \quad \text{clash} \quad \neg A \text{ tagged with mit } \emptyset
\end{align*}
\]

\[
\bullet \quad \text{tag}(A) \cup \text{tag}(\neg A) = \emptyset
\]
Dependency-Directed Backtracking

Example

\((C_1 \sqcup D_1) \sqcap \ldots \sqcap (C_n \sqcup D_n) \sqcap \exists r. \neg A \sqcap \forall r. A \in L(v)\)  
tagged with \(\emptyset\)

\[L(v) := \begin{cases} 
L(v) \cup \{(C_1 \sqcup D_1), \ldots, (C_n \sqcup D_n), 
\exists r. \neg A, \forall r. A\} & \text{all with } \emptyset \\
L(v) \cup \{C_1\} & \text{tagged with } \{1\} \\
\vdots & \vdots & \vdots \\
L(v) \cup \{C_n\} & \text{tagged with } \{n\} \\
\end{cases}\]

\[L(w) := \begin{cases} 
\neg A & \text{tagged with mit } \emptyset \\
\{\neg A\} & \text{tagged with } \emptyset \\
\{\neg A, A\} & \text{clash} \\
\end{cases}\]

- \(\text{tag}(A) \cup \text{tag}(\neg A) = \emptyset\)
- None of the \(\sqcup\)-rules has contributed to the contradiction
Dependency-Directed Backtracking

Example

\[
(C_1 \sqcup D_1) \cap \ldots \cap (C_n \sqcup D_n) \cap \exists r. \neg A \cap \forall r. A \in L(v) \quad \text{tagged with } \emptyset
\]

\[
\sqcap -\text{rule} \quad L(v) := L(v) \cup \{(C_1 \sqcup D_1), \ldots, (C_n \sqcup D_n),
\exists r. \neg A, \forall r. A\} \quad \text{all with } \emptyset
\]

\[
\sqcup -\text{rule} \quad L(v) := L(v) \cup \{C_1\}
\]

\[
\exists -\text{rule} \quad L(w) := \{\neg A\}
\]

\[
\forall -\text{rule} \quad L(w) := \{\neg A, A\} \quad \text{clash}
\]

\[
\text{mit } \neg A \text{ tagged with } \emptyset
\]

\[
\text{tag}(A) \cup \text{tag}(\neg A) = \emptyset
\]

\[
\text{None of the } \sqcap \text{-rules has contributed to the cotradiction}
\]

\[
\text{Output } \text{false} \text{ (unsatisfiable)}
\]
Agenda

- Recap Tableau Calculus
- Optimizations
  - Unfolding
  - Absorption
  - Dependency-Directed Backtracking
  - Further Optimizations
- Classification
- Summary
Further Optimizations

- Simplification and Normalization
  - quick recognition of trivial contradictions
  - normalization, z.B., \( A \cap (B \cap C) \equiv \cap\{A, B, C\} \), \( \forall r. C \equiv \neg \exists r. \neg C \)
  - simplification, e.g., \( \cap\{A, \ldots, \neg A, \ldots\} \equiv \perp \), \( \exists r. \perp \equiv \perp \), \( \forall r. \top \equiv \top \)
Further Optimizations

- **Simplification and Normalization**
  - quick recognition of trivial contradictions
  - normalization, z.B., $A \cap (B \cap C) \equiv \cap \{A, B, C\}$, $\forall r. C \equiv \neg \exists r. \neg C$
  - simplification, e.g., $\cap \{A, \ldots, \neg A, \ldots\} \equiv \bot$, $\exists r. \bot \equiv \bot$, $\forall r. \top \equiv \top$

- **caching**
  - prevents the repeated construction of equal subtrees
  - $L(v)$ initialized with $\{C_1, \ldots, C_n\}$ via $\exists$- and $\forall$-rules
  - check if satisfiability status is cached, otherwise
  - check satisfiability of $C_1 \cap \ldots \cap C_n$, update the cache
Further Optimizations

• Simplification and Normalization
  – quick recognition of trivial contradictions
  – normalization, z.B., \( A \cap (B \cap C) \equiv \cap \{A, B, C\} \), \( \forall r. C \equiv \neg \exists r. \neg C \)
  – simplification, e.g., \( \cap \{A, \ldots, \neg A, \ldots\} \equiv \bot \), \( \exists r. \bot \equiv \bot \), \( \forall r. \top \equiv \top \)

• caching
  – prevents the repeated construction of equal subtrees
  – \( L(v) \) initialized with \( \{C_1, \ldots, C_n\} \) via \( \exists \)- and \( \forall \)-rules
  – check if satisfiability status is cached, otherwise
  – check satisfiability of \( C_1 \cap \ldots \cap C_n \), update the cache

• heuristics
  – try to find good orders for the “don’t care” nondeterminism
  – e.g., \( \cap, \forall, \lor, \exists \)
Further Optimizations

- **Simplification and Normalization**
  - quick recognition of trivial contradictions
  - normalization, z.B., $A \cap (B \cap C) \equiv \cap \{A, B, C\}$, $\forall r. C \equiv \neg \exists r. \neg C$
  - simplification, e.g., $\cap \{A, \ldots, \neg A, \ldots\} \equiv \bot$, $\exists r. \bot \equiv \bot$, $\forall r. \top \equiv \top$

- **caching**
  - prevents the repeated construction of equal subtrees
  - $L(v)$ initialized with $\{C_1, \ldots, C_n\}$ via $\exists$- and $\forall$-rules
  - check if satisfiability status is cached, otherwise
  - check satisfiability of $C_1 \cap \ldots \cap C_n$, update the cache

- **heuristics**
  - try to find good orders for the “don’t care” nondeterminism
  - e.g., $\cap, \forall, \exists, \top, \bot$

- ...
Agenda

• Recap Tableau Calculus
• Optimizations
  – Unfolding
  – Absorption
  – Dependency-Directed Backtracking
  – Further Optimizations
• Classification
• Summary
Optimizing Classification

One of the most wide-spread tasks for automated reasoning is classification

- compute all subclass relationships between atomic concepts in $\mathcal{T}$
Optimizing Classification

One of the most wide-spread tasks for automated reasoning is classification

- compute all subclass relationships between atomic concepts in $T$
- check for $T \models C \sqsubseteq D$ can be reduced to checking satisfiability of $T$
  together with the ABox $(C \cap \neg D)(a)$ (or, equivalently: $C(a), (\neg D)(a)$)
    - $\Rightarrow$ if $\top$ is satisfiable: subsumption does not hold (as we have constructed a counter-model)
    - $\Rightarrow$ if $\top$ is unsatisfiable: subsumption holds (no counter-model exists)
Optimizing Classification

One of the most wide-spread tasks for automated reasoning is classification

- compute all subclass relationships between atomic concepts in $\mathcal{T}$
- check for $\mathcal{T} \models C \sqsubseteq D$ can be reduced to checking satisfiability of $\mathcal{T}$
  together with the ABox $(C \sqcap \neg D)(a)$ (or, equivalently: $C(a), (\neg D)(a)$)
    $\leadsto$ if $\top$ is satisfiable: subsumption does not hold (as we have
    constructed a counter-model)
    $\leadsto$ if $\top$ is unsatisfiable: subsumption holds (no counter-model exists)
- naïve approach needs $n^2$ subsumption checks for $n$ concept names
- normally cached in the concept hierarchy graph
Concept Hierarchy Graph

TU Dresden

Deduction Systems
Optimizing Classification

most wide-spread technique is called enhanced traversal
Optimizing Classification

most wide-spread technique is called **enhanced traversal**

- hierarchy is created incrementally by introducing concept after concept
Optimizing Classification

most wide-spread technique is called enhanced traversal

- hierarchy is created incrementally by introducing concept after concept
- top-down phase: recognize direct superconcepts
- bottom-up phase: recognize direct subconcepts
most wide-spread technique is called enhanced traversal

- hierarchy is created incrementally by introducing concept after concept
- top-down phase: recognize direct superconcepts
- bottom-up phase: recognize direct subconcepts
- transitivity of $\sqsubseteq$ used to save checks

If $A \sqsubseteq B$ and $C \sqsubseteq D$ hold,
then $B \sqsubseteq C \rightarrow A \sqsubseteq D$
and $A \nsubseteq D \rightarrow B \nsubseteq C$
Enhanced Traversal Example

already created hierarchy:

Goal: insertion of JointDisease

Top-Down Phase:

Bottom-Up Phase:
Enhanced Traversal Example

already created hierarchy:

\[
\top \rightarrow \text{Disease} \rightarrow \text{JointDisease} \rightarrow \text{Arthritis} \rightarrow \text{JuvArthritis} \\
\]

Goal: insertion of JointDisease

Top-Down Phase:

- JointDisease $\sqsubseteq ? \text{ Disease}$

Bottom-Up Phase:

- JuvArthritis $\sqsubseteq \text{JointDisease}$
- JuvDisease $\not\sqsubseteq \text{JointDisease}$
- Arthritis $\sqsubseteq \text{JointDisease}$

TU Dresden Deduction Systems
Enhanced Traversal Example

already created hierarchy:

Goal: insertion of JointDisease

Top-Down Phase:

- JointDisease $\sqsubseteq$ Disease
- JointDisease $\sqsubseteq$? JuvDisease

Bottom-Up Phase:
Enhanced Traversal Example

already created hierarchy:

Goal: insertion of JointDisease

Top-Down Phase:

- JointDisease $\sqsubseteq$ Disease
- JointDisease $\not\sqsubseteq$ JuvDisease
- JointDisease $\sqsubseteq?$ Arthritis

Bottom-Up Phase:
Enhanced Traversal Example

already created hierarchy:

Goal: insertion of JointDisease

Top-Down Phase:

- JointDisease \sqsubseteq \text{Disease}
- JointDisease \not\sqsubseteq \text{JuvDisease}
- JointDisease \not\sqsubseteq \text{Arthritis}
- JointDisease \sqsubseteq \text{Joint}

Bottom-Up Phase:
Enhanced Traversal Example

already created hierarchy:

(⊤)

Disease

Joint

JuvDisease

JointDisease

Arthritis

JuvArthritis

Goal: insertion of JointDisease

Top-Down Phase:

• JointDisease ⊑ Disease
• JointDisease ̸⊑ JuvDisease
• JointDisease ̸⊑ Arthritis
• JointDisease ̸⊑ Joint

Bottom-Up Phase:

• JuvArthritis ⊑ JointDisease
Enhanced Traversal Example

already created hierarchy:

- \( \top \)
- Disease
- Joint
- JuvDisease
- JointDisease
- Arthritis
- JuvArthritis

Goal: insertion of JointDisease

Top-Down Phase:
- JointDisease \( \sqsubseteq \) Disease
- JointDisease \( \not\sqsubseteq \) JuvDisease
- JointDisease \( \not\sqsubseteq \) Arthritis
- JointDisease \( \not\sqsubseteq \) Joint

Bottom-Up Phase:
- JuvArthritis \( \sqsubseteq \) JointDisease
- JuvDisease \( \sqsubseteq ? \) JointDisease
Enhanced Traversal Example

already created hierarchy:

```
⊤
|
+v-- Disease
|    |
|    +-- JointDisease
|       |
|       +-- Arthritis
|           |
|           +-- JuvArthritis
|
```

Goal: insertion of JointDisease

Top-Down Phase:

- JointDisease ⊑ Disease
- JointDisease ⊲ JuvDisease
- JointDisease ⊲ Arthritis
- JointDisease ⊲ Joint

Bottom-Up Phase:

- JuvArthritis ⊑ JointDisease
- JuvDisease ⊲ JointDisease
- Arthritis ⊲ JointDisease
Enhanced Traversal Example

already created hierarchy:

\[
\begin{align*}
\top & \quad \text{Disease} & \quad \text{Joint} \\
\text{JuvDisease} & \quad \text{JointDisease} & \quad \text{Arthritis} \\
\text{Arthritis} & \quad & \\
\text{JuvArthritis} & \quad & \\
\end{align*}
\]

Goal: insertion of JointDisease

Top-Down Phase:

- JointDisease $\subseteq$ Disease
- JointDisease $\not\subseteq$ JuvDisease
- JointDisease $\not\subseteq$ Arthritis
- JointDisease $\not\subseteq$ Joint

Bottom-Up Phase:

- JuvArthritis $\subseteq$ JointDisease
- JuvDisease $\not\subseteq$ JointDisease
- Arthritis $\subseteq$ JointDisease
Agenda

- Recap Tableau Calculus
- Optimizations
  - Unfolding
  - Absorption
  - Dependency-Directed Backtracking
  - Further Optimizations
- Classification
- Summary
Summary

- we have a tableau algorithm for $\text{ALCIF}$ knowledge bases
  - ABox treated like for $\text{ALC}$
  - number restrictions are treated similar to functionality and existential quantifiers
- termination via cycle detection
  - becomes harder as the logic becomes more expressive
- naive tableau algorithm not sufficiently performant
- diverse optimizations improve average case
- specific methods for classification
  - enhanced traversal
- tableaux algorithms or variants modifications thereof are the basis of OWL reasoners