DEDUCTION SYSTEMS

Optimizations for Tableau Procedures

Sebastian Rudolph
Agenda

- **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
  - $T$-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

- 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 $\mathcal{T}$

$$\mathcal{T}:$$

$$A \square B \sqcap \exists r.C$$

$$B \equiv C \sqcup D$$

$$C \sqsubseteq \exists r.D$$
Unfolding Example

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

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

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

\[
\begin{align*}
A \\
\neg A \sqcap B \sqcap \exists r.C
\end{align*}
\]

\( \mathcal{T} \):

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

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

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

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

$$
\begin{align*}
A & \sqsubseteq B \sqcap \exists r.C \\
\sim & \sim A \sqcap B \sqcap \exists r.C \\
\sim & \sim A \sqcap (C \sqcup D) \sqcap \exists r.C \\
\sim & \sim A \sqcap ((C \sqcap \exists r.D) \sqcup D) \sqcap \exists r.(C \sqcap \exists r.D)
\end{align*}
$$

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

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

$$\mathcal{T}:
\begin{align*}
A & \subseteq B \sqcap \exists r.C \\
A & \sqsupseteq A \sqcap B \sqcap \exists r.C \\
A & \sqsupseteq A \sqcap (C \sqcup D) \sqcap \exists r.C \\
A & \sqsupseteq A \sqcap ((C \sqcap \exists r.D) \sqcup D) \sqcap \exists r.(C \sqcap \exists r.D)
\end{align*}$$

• $A$ 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
Tableau Algorithm Example with Unfolding

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

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

We obtain the following contradiction-free tableau for the satisfiability of
\[ U = A \land ((C \land \exists r.D) \lor D) \land \exists r.(C \land \exists r.D) : \]

\begin{align*}
L(v_0) &= \{ U, A, (C \land \exists r.D) \lor D, \\
&\quad \exists r.(C \land \exists r.D), C \land \exists r.D, \\
&\quad C, \exists r.D \} \\
L(v_1) &= \{ C \land \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 $C \sqcap \neg C$ 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: $C \sqcap A \sqcap B \sqcap (\neg C \sqcup \neg A \sqcup \neg B)$

- better: apply NNF and unfolding if needed, via corresponding tableau rules:
  - $A \equiv C \Rightarrow A \sqsubseteq C$ 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 NNF$(\neg C) \notin L(v)$,
  let $L(v) := L(v) \cup \{\text{NNF}(\neg C)\}$.
Agenda

• 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 s, 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., \( A \sqsubseteq B \sqcup \neg C \).
2. Transform the axiom:
   - If \( \mathcal{T}_u \) contains an axiom of the form \( A \equiv D \) (\( A \sqsubseteq D \) and \( D \sqsubseteq A \)), then \( A \sqsubseteq C \sqcup \neg B \) cannot be absorbed; \( A \sqsubseteq C \sqcup \neg B \) remains in \( \mathcal{T}_g \).
3. Otherwise, if \( \mathcal{T}_u \) contains an axiom of the form \( A \sqsubseteq D \), absorb \( A \sqsubseteq C \sqcup \neg B \) resulting in \( A \sqsubseteq D \sqcap \left( C \sqcup \neg B \right) \).
4. Otherwise move \( A \sqsubseteq C \sqcup \neg B \) to \( \mathcal{T}_u \).

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

- Nondeterministic: \( B \sqsubseteq C \sqcup \neg A \) also possible.
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., \( A \sqcap B \sqsubseteq C \)
  2. transform the axiom: \( A \sqsubseteq C \sqcup \neg B \)
  3. if \( \mathcal{T}_u \) contains an axiom of the form \( A \equiv D \) \( (A \sqsubseteq D \text{ and } D \sqsupseteq A) \),
     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 $\mathcal{T}_g$ (GCI, not unfoldable)
  – $\mathcal{T}_u$ is treated via $\sqsubseteq$- and $\sqsupseteq$-rules
  – $\mathcal{T}_g$ is treated via the $\sqsubseteq$-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 $\mathcal{T}_u$

• If $A \equiv D \in \mathcal{T}_u$, try rewriting/absorption with other axioms in $\mathcal{T}_u$
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., $A \sqcap B \sqsubseteq C$
  2. transform the axiom: $A \sqsubseteq C \sqcup \neg B$
  3. if $\mathcal{T}_u$ contains an axiom of the form $A \equiv D$ (if $A \sqsubseteq D$ 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 $\mathcal{T}_u$

- nondeterministic: $B \sqsubseteq C \sqcup \neg A$ also possible
Agenda

- 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 $(C_1 \cup D_1) \cap \ldots \cap (C_n \cup D_n) \cap \exists r. \neg A \cap \forall r. A \in L(v)$
Dependency-Directed Backtracking

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

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

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

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

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

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

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

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

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

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

\[
\begin{align*}
\sqcup \text{-rule } L(v) & := \ L(v) \cup \{C_n\} \\
\exists \text{-rule } L(w) & := \ \{\neg A\} \\
\forall \text{-rule } L(w) & := \ \{\neg A, A\} \quad \text{clash}
\end{align*}
\]
• despite those optimizations, search space often to big
• let \( v \in V \) with \( (C_1 \sqcup D_1) \sqcap \ldots \sqcap (C_n \sqcup D_n) \sqcap \exists r. \neg A \sqcap \forall r. A \in L(v) \)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

\[
\begin{align*}
\sqcap \text{-rule } L(v) & := L(v) \cup \{C_n\} \\
\exists \text{-rule } L(w) & := \{-A\} \\
\forall \text{-rule } L(w) & := \{-A, A\} \quad \text{clash} \\
\sqcup \text{-rule } L(v) & := L(v) \cup \{D_n\} \\
\exists \text{-rule } L(w) & := \{-A\} \\
\forall \text{-rule } L(w) & := \{-A, A\} \quad \text{clash} \\
\end{align*}
\]

- exponentially big search space is traversed
Dependency-Directed Backtracking

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

\[ \text{backjumping works roughly as follows:} \]
- 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

- 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 ∅
  – 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
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

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

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

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

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

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

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

Example

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

\begin{align*}
\text{\textbf{\textless}} & \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\}\quad \text{all with } \emptyset \\
\text{\textbf{\textgreater}} & \text{-rule } L(v) := L(v) \cup \{C_1\}\quad \text{C}_1 \text{ tagged with } \{1\} \\
\quad & \quad \vdots \quad \vdots \quad \vdots \\
\text{\textbf{\textless}} & \text{-rule } L(v) := L(v) \cup \{C_n\}\quad \text{C}_n \text{ tagged with } \{n\} \\
\exists & \text{-rule } L(w) := \{\neg A\}\quad A, r \text{ tagged with } \emptyset
\end{align*}

TU Dresden Deduction Systems
Dependency-Directed Backtracking

Example

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

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

\[
\begin{align*}
\forall \text{-rule} & \quad L(v) := L(v) \cup \{C_1\} \quad C_1 \text{ tagged with } \{1\}
\end{align*}
\]

\[
\begin{align*}
\forall \text{-rule} & \quad L(v) := L(v) \cup \{C_n\} \quad C_n \text{ tagged with } \{n\}
\end{align*}
\]

\[
\begin{align*}
\exists \text{-rule} & \quad L(w) := \{-A\} \quad A, r \text{ tagged with } \emptyset
\end{align*}
\]

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

Example

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

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

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

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

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

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

Example

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

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

• \(\text{tag}(A) \cup \text{tag}(\neg A) = \emptyset\)
Dependency-Directed Backtracking

Example

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

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

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

Example

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

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

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

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

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

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

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

- \(\text{tag}(A) \cup \text{tag}(\neg A) = \emptyset\)
- None of the \(\sqcup\)-rules has contributed to the cotractiction
- Output false (unsatisfiable)
Agenda

- 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., $\cap\{A, \ldots, \neg A, \ldots\} \equiv \bot$, $\exists r. \bot \equiv \bot$, $\forall r. \top \equiv \top$
Further Optimizations

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

- **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 $C_1 \land \ldots \land C_n$, update the cache
Further Optimizations

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

- 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 $C_1 \land \ldots \land C_n$, update the cache

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

- ...
Agenda

• 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 $\mathcal{T}$
- 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: $C(a), (\neg D)(a)$)
  - $\Rightarrow$ 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

- ⊤
  - Disease
    - JuvDisease
      - Arthritis
        - JuvArthritis
  - Joint
    - JointDisease

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

\[
\text{If } A \sqsubseteq B \text{ and } C \sqsubseteq D \text{ hold, then } B \sqsubseteq C \implies A \sqsubseteq D
\]

\[
A \not\sqsubseteq D \implies B \not\sqsubseteq C
\]
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
- 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 $A \not\sqsubseteq D \rightarrow B \not\sqsubseteq C$
**Enhanced Traversal Example**

**already created hierarchy:**

- $\top$
- **Disease**
  - **JuvDisease**
  - **JointDisease**
  - **Arthritis**
  - **Joint**

**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** $\not\sqsubseteq$ **JointDisease**
- **Arthritis** $\sqsubseteq$ **JointDisease**

**TU Dresden Deduction Systems**
Enhanced Traversal Example

already created hierarchy:

\[ \top \rightarrow \text{Disease} \rightarrow \text{Joint} \rightarrow \text{JuvDisease} \rightarrow \text{JointDisease} \rightarrow \text{Arthritis} \rightarrow \text{JuvArthritis} \] is a subgraph of \[ \top \rightarrow \text{Disease} \rightarrow \text{Joint} \rightarrow \text{JuvDisease} \rightarrow \text{JointDisease} \rightarrow \text{Arthritis} \rightarrow \text{JuvArthritis} \]

Goal: insertion of JointDisease

Top-Down Phase:
- JointDisease \( \subseteq \top \text{ Disease} \)

Bottom-Up Phase:
- JuvArthritis \( \subseteq \text{JointDisease} \)
- JuvDisease \( \nsubseteq \text{JointDisease} \)
- Arthritis \( \subseteq \text{JointDisease} \)
already created hierarchy:

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

Goal: insertion of JointDisease

Top-Down Phase:

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

Bottom-Up Phase:

- JuvArthritis $\sqsubseteq$ JointDisease
- JuvDisease $\nless$ JointDisease
- Arthritis $\sqsubseteq$ JointDisease
**Enhanced Traversal Example**

**already created hierarchy:**

```
⊤
├── Disease
│   ├── JuvDisease
│   │   └── Arthritis
│   └── JointDisease
│       └── Joint
│           └── Arthritis
│                   └── JuvArthritis
└── JointDisease
```

**Goal: insertion of JointDisease**

**Top-Down Phase:**
- JointDisease ⊑ Disease
- JointDisease ⊑ Arthritis
- JointDisease ⊑ ? Arthritis

**Bottom-Up Phase:**
- JuvArthritis ⊑ JointDisease
- JuvDisease ⊑ JointDisease
- Arthritis ⊑ JointDisease
Enhanced Traversal Example

already created hierarchy:

Goal: insertion of JointDisease

Top-Down Phase:
- JointDisease $\sqsupseteq$ Disease
- JointDisease $\not\sqsubseteq$ JuvDisease
- JointDisease $\not\sqsubseteq$ Arthritis
- JointDisease $\sqsubseteq?\ Joint$

Bottom-Up Phase:
Enhanced Traversal Example

already created hierarchy:

```
⊤
  ┌─ Disease
  │   └─ Joint
  │       │
  │       └─ JointDisease
  │           └─ JuvDisease
  │               └─ 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:

\[
\begin{array}{c}
\top \\
\downarrow \\
\text{Disease} \\
\downarrow \\
\text{JuvDisease} \\
\downarrow \\
\text{JointDisease} \\
\downarrow \\
\text{Arthritis} \\
\downarrow \\
\text{JuvArthritis} \\
\end{array}
\]

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:

\[
\begin{array}{c}
\top \\
\text{Disease} \\
\text{Joint} \\
\text{JuvDisease} \\
\text{JointDisease} \\
\text{Arthritis} \\
\text{JuvArthritis} \\
\text{Joint} \\
\end{array}
\]

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:

\[
\begin{array}{c}
\top \\
\downarrow \\
\text{Disease} \\
\downarrow \\
\text{JuvDisease} \\
\downarrow \\
\text{JointDisease} \\
\downarrow \\
\text{Arthritis} \\
\downarrow \\
\text{JuvArthritis}
\end{array}
\]

Goal: insertion of JointDisease

Top-Down Phase:

- JointDisease ⊑ Disease
- JointDisease ⊏ JuvDisease
- JointDisease ⊏ Arthritis
- JointDisease ⊏ Joint

Bottom-Up Phase:

- JuvArthritis ⊏ JointDisease
- JuvDisease ⊏ JointDisease
- Arthritis ⊏ JointDisease

TU Dresden Deduction Systems
Agenda

- Optimizations
  - Unfolding
  - Absorption
  - Dependency-Directed Backtracking
  - Further Optimizations
- Classification
- Summary
Summary

- we have a tableau algorithm for $ALCIF$ knowledge bases
  - ABox treated like for $ALC$
  - 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