Implementing DL Systems
Naive Implementations

Problems include:

Naive Implementations

Problems include:

☞ Space usage

⤷ Storage required for tableaux datastructures
(DIR)Rarely a serious problem in practice

☞ Time usage
(DIR)Search required due to non-deterministic expansion
(DIR)Serious problem in practice

Mitigated by:

➙ Careful choice of algorithm
➙ Highly optimised implementation
Naive Implementations

Problems include:

☞ Space usage
  • Storage required for tableaux datastructures
Naive Implementations

Problems include:

- Space usage
  - Storage required for tableaux datastructures
  - Rarely a serious problem in practice

- Time usage
  - Search required due to non-deterministic expansion
    - Serious problem in practice
    - Mitigated by:
      - Careful choice of algorithm
      - Highly optimised implementation
Naive Implementations

Problems include:

☞ Space usage
  - Storage required for tableaux datastructures
  - Rarely a serious problem in practice

☞ Time usage
Naive Implementations

Problems include:

☞ Space usage
  • Storage required for tableaux datastructures
  • Rarely a serious problem in practice

☞ Time usage
  • Search required due to non-deterministic expansion
Naive Implementations

Problems include:

☞ **Space usage**
  - Storage required for tableaux datastructures
  - Rarely a serious problem in practice

☞ **Time usage**
  - Search required due to non-deterministic expansion
  - **Serious** problem in practice
Naive Implementations

Problems include:

- **Space usage**
  - Storage required for tableaux datastructures
  - Rarely a serious problem in practice

- **Time usage**
  - Search required due to non-deterministic expansion
  - **Serious** problem in practice
  - Mitigated by:
Naive Implementations

Problems include:

☞ Space usage
  - Storage required for tableaux datastructures
  - Rarely a serious problem in practice

☞ Time usage
  - Search required due to non-deterministic expansion
  - **Serious** problem in practice
  - Mitigated by:
    ➞ Careful *choice of algorithm*
Naive Implementations

Problems include:

ząd Space usage
  • Storage required for tableaux datastructures
  • Rarely a serious problem in practice

czą Time usage
  • Search required due to non-deterministic expansion
  • **Serious** problem in practice
  • Mitigated by:
    ➔ Careful *choice of algorithm*
    ➔ Highly *optimised implementation*
Careful Choice of Algorithm

- Transitive roles instead of transitive closure
- Deterministic expansion of \( R : C \)
- (Relatively) simple blocking conditions
- Cycles always represent (part of) cyclical models

- Direct algorithm/implementation instead of encodings
- GCI axioms can be used to "encode" additional operators/axioms
  - Powerful technique, particularly when used with FL closure
  - Can encode cardinality constraints, inverse roles, range/domain, ...
  - E.g., (domain \( R : C \)) > v C

- FL encodings introduce (large numbers of) axioms
  - BUT even simple domain encoding is disastrous with large numbers of roles
Careful Choice of Algorithm

Transitive roles instead of transitive closure
Careful Choice of Algorithm

- Transitive roles instead of transitive closure
  - Deterministic expansion of $\exists R.C$, even when $R \in R_+$
Careful Choice of Algorithm

- Transitive roles instead of transitive closure
  - Deterministic expansion of $\exists R.C$, even when $R \in \mathbb{R}_+$
  - (Relatively) simple blocking conditions
Careful Choice of Algorithm

Transitive roles instead of transitive closure

- Deterministic expansion of $\exists R.C$, even when $R \in \mathbb{R}_+$
- (Relatively) simple blocking conditions
- Cycles **always** represent (part of) cyclical models
Careful Choice of Algorithm

- Transitive roles instead of transitive closure
  - Deterministic expansion of $\exists R.C$, even when $R \in \mathbb{R}_+^+$
  - (Relatively) simple blocking conditions
  - Cycles always represent (part of) cyclical models
- Direct algorithm/implementation instead of encodings

- GCI axioms can be used to "encode" additional operators/axioms
  - Powerful technique, particularly when used with FL closure
  - Can encode cardinality constraints, inverse roles, range/domain, etc.
  - E.g., $(\text{domain } R : C) \land R > v C$
  - (FL) encodings introduce (large numbers of) axioms
  - BUT even simple domain encoding is disastrous with large numbers of roles
Transitive roles instead of transitive closure
- Deterministic expansion of $\exists R.C$, even when $R \in R_+$
- (Relatively) simple blocking conditions
- Cycles **always** represent (part of) cyclical models

Direct algorithm/implementation instead of encodings
- GCI axioms can be used to “encode” additional operators/axioms
Careful Choice of Algorithm

- Transitive roles instead of transitive closure
  - Deterministic expansion of $\exists R.C$, even when $R \in \mathbb{R}_+$
  - (Relatively) simple blocking conditions
  - Cycles **always** represent (part of) cyclical models

- Direct algorithm/implementation instead of encodings
  - GCI axioms can be used to “encode” additional operators/axioms
  - Powerful technique, particularly when used with FL closure
Careful Choice of Algorithm

- Transitive roles instead of transitive closure
  - Deterministic expansion of $\exists R.C$, even when $R \in \mathbb{R}_+$
  - (Relatively) simple blocking conditions
  - Cycles **always** represent (part of) cyclical models

- Direct algorithm/implementation instead of encodings
  - GCI axioms can be used to “encode” additional operators/axioms
  - Powerful technique, particularly when used with FL closure
  - Can encode cardinality constraints, inverse roles, range/domain, ...

E.g., $(\text{domain } R:C) \text{9} R: > v C$ (FL) encodings introduce (large numbers of) axioms

BUT even simple domain encoding is disastrous with large numbers of roles
Careful Choice of Algorithm

- Transitive roles instead of transitive closure
  - Deterministic expansion of $\exists R.C$, even when $R \in R_+$
  - (Relatively) simple blocking conditions
  - Cycles **always** represent (part of) cyclical models

- Direct algorithm/implementation instead of encodings
  - GCI axioms can be used to “encode” additional operators/axioms
  - Powerful technique, particularly when used with FL closure
  - Can encode cardinality constraints, inverse roles, range/domain,

  \[ \rightarrow \text{E.g., } (\text{domain } R.C) \equiv \exists R. \top \subseteq C \]
Careful Choice of Algorithm

- Transitive roles instead of transitive closure
  - Deterministic expansion of $\exists R. C$, even when $R \in \mathbb{R}_+$
  - (Relatively) simple blocking conditions
  - Cycles **always** represent (part of) cyclical models

- Direct algorithm/implementation instead of encodings
  - GCI axioms can be used to “encode” additional operators/axioms
  - Powerful technique, particularly when used with FL closure
  - Can encode cardinality constraints, inverse roles, range/domain, ...

  $\rightarrow$ E.g., $(\text{domain } R.C) \equiv \exists R. \top \subseteq C$

  - (FL) encodings introduce (large numbers of) axioms
Careful Choice of Algorithm

- Transitive roles instead of transitive closure
  - Deterministic expansion of $\exists R.C$, even when $R \in R_+$
  - (Relatively) simple blocking conditions
  - Cycles **always** represent (part of) cyclical models

- Direct algorithm/implementation instead of encodings
  - GCI axioms can be used to “encode” additional operators/axioms
  - Powerful technique, particularly when used with FL closure
  - Can encode cardinality constraints, inverse roles, range/domain,
    ...
    ➔ E.g., $(\text{domain } R.C) \equiv \exists R.\top \subseteq C$
  - (FL) encodings introduce (large numbers of) axioms
  - **BUT** even simple domain encoding is **disastrous** with large numbers of roles
Highly Optimised Implementation

Optimisation performed at 2 levels
Highly Optimised Implementation

Optimisation performed at 2 levels

☞ Computing **classification** (partial ordering) of concepts
Highly Optimised Implementation

Optimisation performed at 2 levels

☞ Computing **classification** (partial ordering) of concepts
  
  ● Objective is to minimise number of subsumption tests

Can use standard order-theoretic techniques ➜ E.g., use enhanced traversal that exploits information from previous tests

Also use structural information from KB ➜ E.g., to select order in which to classify concepts

Computing **subsumption** between concepts

Objective is to minimise cost of single subsumption tests

Small number of hard tests can dominate classification time

Recent DL research has addressed this problem (with considerable success)
Highly Optimised Implementation

Optimisation performed at 2 levels

Computing **classification** (partial ordering) of concepts

- Objective is to minimise number of subsumption tests
- Can use standard order-theoretic techniques
Highly Optimised Implementation

Optimisation performed at 2 levels

☞ Computing **classification** (partial ordering) of concepts
  - Objective is to minimise number of subsumption tests
  - Can use standard order-theoretic techniques
    ➔ E.g., use **enhanced traversal** that exploits information from previous tests

Optimisation performed at 2 levels

☞ Computing **classification** (partial ordering) of concepts
  - Objective is to minimise number of subsumption tests
  - Can use standard order-theoretic techniques
    ➔ E.g., use **enhanced traversal** that exploits information from previous tests
Highly Optimised Implementation

Optimisation performed at 2 levels

Computing **classification** (partial ordering) of concepts
  - Objective is to minimise number of subsumption tests
  - Can use standard order-theoretic techniques
    - E.g., use **enhanced traversal** that exploits information from previous tests
  - Also use structural information from KB
Highly Optimised Implementation

Optimisation performed at 2 levels

Computing classification (partial ordering) of concepts
- Objective is to minimise number of subsumption tests
- Can use standard order-theoretic techniques
  - E.g., use enhanced traversal that exploits information from previous tests
- Also use structural information from KB
  - E.g., to select order in which to classify concepts

Computing subsumption between concepts
- Objective is to minimise cost of single subsumption tests
- Small number of hard tests can dominate classification time
- Recent DL research has addressed this problem (with considerable success)
Highly Optimised Implementation

Optimisation performed at 2 levels

 Computing **classification** (partial ordering) of concepts
  - Objective is to minimise number of subsumption tests
  - Can use standard order-theoretic techniques
    ➔ E.g., use **enhanced traversal** that exploits information from previous tests
  - Also use structural information from KB
    ➔ E.g., to select order in which to classify concepts

 Computing **subsumption** between concepts
Highly Optimised Implementation

Optimisation performed at 2 levels

Computing **classification** (partial ordering) of concepts
- Objective is to minimise number of subsumption tests
- Can use standard order-theoretic techniques
  - E.g., use **enhanced traversal** that exploits information from previous tests
- Also use structural information from KB
  - E.g., to select order in which to classify concepts

Computing **subsumption** between concepts
- Objective is to minimise cost of single subsumption tests
Highly Optimised Implementation

Optimisation performed at 2 levels

Computing **classification** (partial ordering) of concepts
- Objective is to minimise number of subsumption tests
- Can use standard order-theoretic techniques
  - E.g., use **enhanced traversal** that exploits information from previous tests
- Also use structural information from KB
  - E.g., to select order in which to classify concepts

Computing **subsumption** between concepts
- Objective is to minimise cost of single subsumption tests
- Small number of hard tests can dominate classification time
Highly Optimised Implementation

Optimisation performed at 2 levels

Computing **classification** (partial ordering) of concepts
- Objective is to minimise number of subsumption tests
- Can use standard order-theoretic techniques
  - E.g., use **enhanced traversal** that exploits information from previous tests
- Also use structural information from KB
  - E.g., to select order in which to classify concepts

Computing **subsumption** between concepts
- Objective is to minimise cost of single subsumption tests
- Small number of hard tests can dominate classification time
- Recent DL research has addressed this problem (with considerable success)
Optimising Subsumption Testing

Optimisation techniques broadly fall into 2 categories:

- **Pre-processing optimisations**: Aim is to simplify KB and facilitate subsumption testing. Largely algorithm independent. Particularly important when KB contains GCI axioms.

- **Algorithmic optimisations**: Main aim is to reduce search space due to non-determinism. Integral part of implementation. But often generally applicable to search based algorithms.
Optimising Subsumption Testing

Optimisation techniques broadly fall into 2 categories

- Pre-processing optimisations

Aim is to simplify KB and facilitate subsumption testing. Largely algorithm independent. Particularly important when KB contains GCI axioms.

- Algorithmic optimisations

Main aim is to reduce search space due to non-determinism. Integral part of implementation. But often generally applicable to search based algorithms.
Optimising Subsumption Testing

**Optimisation techniques** broadly fall into 2 categories

☞ Pre-processing optimisations
  - Aim is to *simplify KB* and facilitate subsumption testing
Optimising Subsumption Testing

Optimisation techniques broadly fall into 2 categories

- Pre-processing optimisations
  - Aim is to simplify KB and facilitate subsumption testing
  - Largely algorithm independent

- Algorithmic optimisations
  - Main aim is to reduce search space due to non-determinism
  - Integral part of implementation
  - But often generally applicable to search based algorithms
Optimising Subsumption Testing

Optimisation techniques broadly fall into 2 categories

☞ Pre-processing optimisations
  ● Aim is to simplify KB and facilitate subsumption testing
  ● Largely algorithm independent
  ● Particularly important when KB contains GCI axioms
Optimising Subsumption Testing

**Optimisation techniques** broadly fall into 2 categories

- Pre-processing optimisations
  - Aim is to **simplify KB** and facilitate subsumption testing
  - Largely algorithm independent
  - Particularly important when KB contains GCI axioms

- Algorithmic optimisations
Optimising Subsumption Testing

**Optimisation techniques** broadly fall into 2 categories

- **Pre-processing optimisations**
  - Aim is to *simplify KB* and facilitate subsumption testing
  - Largely algorithm independent
  - Particularly important when KB contains GCI axioms

- **Algorithmic optimisations**
  - Main aim is to *reduce search space* due to non-determinism
Optimising Subsumption Testing

Optimisation techniques broadly fall into 2 categories

- Pre-processing optimisations
  - Aim is to simplify KB and facilitate subsumption testing
  - Largely algorithm independent
  - Particularly important when KB contains GCI axioms

- Algorithmic optimisations
  - Main aim is to reduce search space due to non-determinism
  - Integral part of implementation
Optimising Subsumption Testing

Optimisation techniques broadly fall into 2 categories

☞ Pre-processing optimisations
  • Aim is to simplify KB and facilitate subsumption testing
  • Largely algorithm independent
  • Particularly important when KB contains GCI axioms

☞ Algorithmic optimisations
  • Main aim is to reduce search space due to non-determinism
  • Integral part of implementation
  • But often generally applicable to search based algorithms
Pre-processing Optimisations

Useful techniques include

- Normalisation and simplification of concepts
- Refinement of technique first used in KRIS system
- Lexically normalise and simplify all concepts in KB
- Combine with lazy unfolding in tableaux algorithm
- Facilitates early detection of inconsistencies (clashes)

- Absorption (simplification) of general axioms
- Eliminate GCIs by absorbing into "definition" axioms
- Definition axioms efficiently dealt with by lazy expansion

- Avoidance of potentially costly reasoning whenever possible
- Normalisation can discover "obvious" (un)satisfiability
- Structural analysis can discover "obvious" subsumption
Pre-processing Optimisations

Useful techniques include

☞ Normalisation and simplification of concepts

Refinement of technique first used in KRIS system

Lexically normalise and simplify all concepts in KB

Combine with lazy unfolding in tableaux algorithm

Facilitates early detection of inconsistencies (clashes)

☞ Absorption (simplification) of general axioms

Eliminate GCIs by absorbing into “definition” axioms

Definition axioms efficiently dealt with by lazy expansion

☞ Avoidance of potentially costly reasoning whenever possible

Normalisation can discover “obvious” (un)satisfiability

Structural analysis can discover “obvious” subsumption
Pre-processing Optimisations

Useful techniques include

☞ Normalisation and simplification of concepts
  ● Refinement of technique first used in KRIS system

Implementations – p. 6/14
Pre-processing Optimisations

Useful techniques include

☞ Normalisation and simplification of concepts
  - Refinement of technique first used in KRIS system
  - Lexically normalise and simplify all concepts in KB
Pre-processing Optimisations

Useful techniques include

☞ Normalisation and simplification of concepts
  • Refinement of technique first used in KRIS system
  • Lexically normalise and simplify all concepts in KB
  • Combine with lazy unfolding in tableaux algorithm
Pre-processing Optimisations

Useful techniques include

☞ Normalisation and simplification of concepts
  - Refinement of technique first used in KRIS system
  - Lexically normalise and simplify all concepts in KB
  - Combine with lazy unfolding in tableaux algorithm
  - Facilitates early detection of inconsistencies (clashes)
Pre-processing Optimisations

Useful techniques include

☞ Normalisation and simplification of concepts
  - Refinement of technique first used in KRIS system
  - Lexically normalise and simplify all concepts in KB
  - Combine with lazy unfolding in tableaux algorithm
  - Facilitates early detection of inconsistencies (clashes)

☞ Absorption (simplification) of general axioms
Pre-processing Optimisations

Useful techniques include

- Normalisation and simplification of concepts
  - Refinement of technique first used in KRIS system
  - Lexically normalise and simplify all concepts in KB
  - Combine with lazy unfolding in tableaux algorithm
  - Facilitates early detection of inconsistencies (clashes)

- Absorption (simplification) of general axioms
  - Eliminate GCIs by absorbing into “definition” axioms
Pre-processing Optimisations

Useful techniques include

☞ Normalisation and simplification of concepts
  - Refinement of technique first used in KRIS system
  - Lexically normalise and simplify all concepts in KB
  - Combine with lazy unfolding in tableaux algorithm
  - Facilitates early detection of inconsistencies (clashes)

☞ Absorption (simplification) of general axioms
  - Eliminate GCIs by absorbing into “definition” axioms
  - Definition axioms efficiently dealt with by lazy expansion
Pre-processing Optimisations

Useful techniques include

☞ Normalisation and simplification of concepts
  • Refinement of technique first used in KRIS system
  • Lexically normalise and simplify all concepts in KB
  • Combine with lazy unfolding in tableaux algorithm
  • Facilitates early detection of inconsistencies (clashes)

☞ Absorption (simplification) of general axioms
  • Eliminate GCIs by absorbing into “definition” axioms
  • Definition axioms efficiently dealt with by lazy expansion

☞ Avoidance of potentially costly reasoning whenever possible
Pre-processing Optimisations

Useful techniques include

☞ Normalisation and simplification of concepts
  - Refinement of technique first used in KRIS system
  - Lexically normalise and simplify all concepts in KB
  - Combine with lazy unfolding in tableaux algorithm
  - Facilitates early detection of inconsistencies (clashes)

☞ Absorption (simplification) of general axioms
  - Eliminate GCIs by absorbing into “definition” axioms
  - Definition axioms efficiently dealt with by lazy expansion

☞ Avoidance of potentially costly reasoning whenever possible
  - Normalisation can discover “obvious” (un)satisfiability
Pre-processing Optimisations

Useful techniques include

☞ Normalisation and simplification of concepts
  - Refinement of technique first used in *Kris* system
  - Lexically normalise and simplify all concepts in KB
  - Combine with lazy unfolding in tableaux algorithm
  - Facilitates early detection of inconsistencies (clashes)

☞ Absorption (simplification) of general axioms
  - Eliminate GCIs by absorbing into “definition” axioms
  - Definition axioms efficiently dealt with by lazy expansion

☞ Avoidance of potentially costly reasoning whenever possible
  - Normalisation can discover “obvious” (un)satisfiability
  - Structural analysis can discover “obvious” subsumption
Normalisation and Simplification

Normalise concepts to standard form, e.g.:

- $9 \mathbb{R} C$
- $\text{Ct} \mathbb{D}!$(
- $\text{Cu} : \mathbb{D}$)

Simplify concepts, e.g.:

- $(\mathbb{D} \text{u} \text{C}) \text{u} (\mathbb{A} \text{u} \mathbb{D})! \mathbb{A} \text{u} \text{C} \text{u} \mathbb{D} ! : :$

Lazily unfold concepts in tableaux algorithm

- Use names/pointers to refer to complex concepts
- Only add structure as required by progress of algorithm
- Detect clashes between lexically equivalent concepts

HappyFather

- has-child: (Doctor t Lawyer)
- has-child: (Doctor u Lawyer)

 Implementation – p. 7/14
Normalisation and Simplification

- Normalise concepts to standard form, e.g.:
Normalisation and Simplification

- Normalise concepts to standard form, e.g.:
  - $\exists R.C \rightarrow \forall R.\neg C$

- Lazily unfold concepts in tableaux algorithm
- Use names/pointers to refer to complex concepts
- Only add structure as required by progress of algorithm
- Detect clashes between lexically equivalent concepts

Implementation – p. 7/14
Normalisation and Simplification

- Normalise concepts to standard form, e.g.:
  - $\exists R.C \rightarrow \forall R. \neg C$
  - $C \sqcup D \rightarrow \neg(\neg C \sqcap \neg D)$
Normalisation and Simplification

- Normalise concepts to standard form, e.g.:
  - $\exists R.C \rightarrow \neg\forall R.\neg C$
  - $C \sqcup D \rightarrow \neg(\neg C \cap \neg D)$

- Simplify concepts, e.g.:
Normalisation and Simplification

☞ Normalise concepts to standard form, e.g.:
  - \( \exists R.C \rightarrow \neg \forall R.\neg C \)
  - \( C \sqcup D \rightarrow \neg (\neg C \sqcap \neg D) \)

☞ Simplify concepts, e.g.:
  - \( (D \sqcap C') \sqcap (A \sqcap D) \rightarrow A \sqcap C \sqcap D \)
Normalisation and Simplification

Normalise concepts to standard form, e.g.:
- $\exists R. C \rightarrow \neg \forall R. \neg C$
- $C \sqcup D \rightarrow \neg (\neg C \cap \neg D)$

Simplify concepts, e.g.:
- $(D \cap C') \cap (A \cap D) \rightarrow A \cap C \cap D$
- $\forall R. \top \rightarrow \top$
Normalisation and Simplification

- **Normalise concepts to standard form, e.g.:**
  - $\exists R. C \rightarrow \neg \forall R. \neg C$
  - $C \sqcup D \rightarrow \neg(\neg C \sqcap \neg D)$

- **Simplify concepts, e.g.:**
  - $(D \sqcap C') \sqcap (A \sqcap D) \rightarrow A \sqcap C \sqcap D$
  - $\forall R. \top \rightarrow \top$
  - $\ldots \sqcap C \sqcap \ldots \sqcap \neg C \sqcap \ldots \rightarrow \bot$
Normalisation and Simplification

- Normalise concepts to standard form, e.g.:
  - $\exists R. C \longrightarrow \neg \forall R. \neg C$
  - $C \sqcup D \longrightarrow \neg (\neg C \sqcap \neg D)$

- Simplify concepts, e.g.:
  - $(D \sqcap C') \sqcap (A \sqcap D) \longrightarrow A \sqcap C \sqcap D$
  - $\forall R. \top \longrightarrow \top$
  - $\ldots \sqcap C \sqcap \ldots \sqcap \neg C \sqcap \ldots \longrightarrow \bot$

- Lazily unfold concepts in tableaux algorithm
Normalisation and Simplification

- **Normalise concepts to standard form, e.g.:**
  - \( \exists R.C \rightarrow \neg \forall R.\neg C \)
  - \( C \sqcup D \rightarrow \neg (\neg C \sqcap \neg D) \)

- **Simplify concepts, e.g.:**
  - \( (D \sqcap C') \sqcap (A \sqcap D) \rightarrow A \sqcap C \sqcap D \)
  - \( \forall R.\top \rightarrow \top \)
  - \( \ldots \sqcap C \sqcap \ldots \sqcap \neg C \sqcap \ldots \rightarrow \bot \)

- **Lazily unfold concepts in tableaux algorithm**
  - Use names/pointers to refer to complex concepts
Normalisation and Simplification

☞ Normalise concepts to standard form, e.g.:
  - \( \exists R.C \rightarrow \forall R.\neg C \)
  - \( C \sqcup D \rightarrow \neg(\neg C \sqcap \neg D) \)

☞ Simplify concepts, e.g.:
  - \( (D \sqcap C') \sqcap (A \sqcap D) \rightarrow A \sqcap C \sqcap D \)
  - \( \forall R.\top \rightarrow \top \)
  - \( \ldots \sqcap C \sqcap \ldots \sqcap \neg C \sqcap \ldots \rightarrow \bot \)

☞ Lazily unfold concepts in tableaux algorithm
  - Use names/pointers to refer to complex concepts
  - Only add structure as required by progress of algorithm
Normalisation and Simplification

- Normalise concepts to standard form, e.g.:
  - $\exists R. C \rightarrow \neg \forall R. \neg C$
  - $C \sqcup D \rightarrow \neg (\neg C \sqcap \neg D)$

- Simplify concepts, e.g.:
  - $(D \sqcap C') \sqcap (A \sqcap D) \rightarrow A \sqcap C' \sqcap D$
  - $\forall R. \top \rightarrow \top$
  - $\ldots \sqcap C \sqcap \ldots \sqcap \neg C \sqcap \ldots \rightarrow \bot$

- Lazily unfold concepts in tableaux algorithm
  - Use names/pointers to refer to complex concepts
  - Only add structure as required by progress of algorithm
  - Detect clashes between lexically equivalent concepts
Normalisation and Simplification

- Normalise concepts to standard form, e.g.:
  - $\exists R.C \rightarrow \neg\forall R.\neg C$
  - $C \sqcap D \rightarrow \neg(\neg C \sqcap \neg D)$

- Simplify concepts, e.g.:
  - $(D \sqcap C) \sqcap (A \sqcap D) \rightarrow A \sqcap C \sqcap D$
  - $\forall R.\top \rightarrow \top$
  - $\ldots \sqcap C \sqcap \ldots \sqcap \neg C \sqcap \ldots \rightarrow \bot$

- Lazily unfold concepts in tableaux algorithm
  - Use names/pointers to refer to complex concepts
  - Only add structure as required by progress of algorithm
  - Detect clashes between lexically equivalent concepts

\{\text{HappyFather, } \neg\text{HappyFather}\} \rightarrow \text{clash}
\{\forall\text{has-child.}(\text{Doctor} \sqcup \text{Lawyer}), \exists\text{has-child.}(\neg\text{Doctor} \sqcap \neg\text{Lawyer})\} \rightarrow \text{search}
Absorption I
Absorption I

Reasoning w.r.t. set of GCI axioms can be very costly
Absorption I

Disproportionate to set of GCI axioms can be very costly

- GCI $C \subseteq D$ adds $D \cup \neg C$ to every node label
Absorption I

Reasoning w.r.t. set of GCI axioms can be very costly

- GCI $C \subseteq D$ adds $D \sqcup \neg C$ to every node label
- Expansion of disjunctions leads to search
Reasoning w.r.t. set of GCI axioms can be very costly
- GCI $C \subseteq D$ adds $D \sqcup \neg C$ to every node label
- Expansion of disjunctions leads to search
- With 10 axioms and 10 nodes search space already $2^{100}$
Reasoning w.r.t. set of GCI axioms can be very costly

- GCI $C \sqsubseteq D$ adds $D \sqcup \neg C$ to every node label
- Expansion of disjunctions leads to search
- With 10 axioms and 10 nodes search space already $2^{100}$
- GALEN (medical terminology) KB contains hundreds of axioms
Absorption I

Reasoning w.r.t. set of GCI axioms can be very costly
- GCI $C \subseteq D$ adds $D \sqcup \neg C$ to every node label
- Expansion of disjunctions leads to search
- With 10 axioms and 10 nodes search space already $2^{100}$
- GALEN (medical terminology) KB contains hundreds of axioms

Reasoning w.r.t. “primitive definition” axioms is relatively efficient
Reaching w.r.t. set of GCI axioms can be very costly

- GCI \( C \subseteq D \) adds \( D \sqcup \neg C \) to every node label
- Expansion of disjunctions leads to search
- With 10 axioms and 10 nodes search space already \( 2^{100} \)
- \textsc{Galen} (medical terminology) KB contains hundreds of axioms

Reasoning w.r.t. “primitive definition” axioms is relatively efficient

- For \( \text{CN} \subseteq D \), add \( D \) only to node labels containing CN

Implementation – p. 8/14
Absorption I

Reasoning w.r.t. set of GCI axioms can be very costly
- GCI \( C \sqsubseteq D \) adds \( D \sqcap \neg C \) to every node label
- Expansion of disjunctions leads to search
- With 10 axioms and 10 nodes search space already \( 2^{100} \)
- GALEN (medical terminology) KB contains hundreds of axioms

Reasoning w.r.t. “primitive definition” axioms is relatively efficient
- For CN \( \sqsubseteq D \), add \( D \) only to node labels containing CN
- For CN \( \sqsupseteq D \), add \( \neg D \) only to node labels containing \( \neg CN \)
Absorption I

Reasoning w.r.t. set of GCI axioms can be very costly
- GCI $C \subseteq D$ adds $D \sqcup \neg C$ to every node label
- Expansion of disjunctions leads to search
- With 10 axioms and 10 nodes search space already $2^{100}$
- GALEN (medical terminology) KB contains hundreds of axioms

Reasoning w.r.t. “primitive definition” axioms is relatively efficient
- For CN $\sqsubseteq D$, add $D$ only to node labels containing CN
- For CN $\sqsupseteq D$, add $\neg D$ only to node labels containing $\neg$CN
- Can expand definitions lazily
Absorption I

Reasoning w.r.t. set of GCI axioms can be very costly
- GCI $C \sqsubseteq D$ adds $D \sqcup \neg C$ to every node label
- Expansion of disjunctions leads to search
- With 10 axioms and 10 nodes search space already $2^{100}$
- GALEN (medical terminology) KB contains hundreds of axioms

Reasoning w.r.t. “primitive definition” axioms is relatively efficient
- For CN $\sqsubseteq D$, add $D$ only to node labels containing CN
- For CN $\sqsupset D$, add $\neg D$ only to node labels containing $\neg$CN
- Can expand definitions lazily
  - Only add definitions after other local (propositional) expansion
Absorption I

- Reasoning w.r.t. set of GCI axioms can be very costly
  - GCI \( C \subseteq D \) adds \( D \sqcup \neg C \) to every node label
  - Expansion of disjunctions leads to search
  - With 10 axioms and 10 nodes search space already \( 2^{100} \)
  - **GALEN** (medical terminology) KB contains **hundreds** of axioms

- Reasoning w.r.t. “primitive definition” axioms is relatively efficient
  - For \( CN \subseteq D \), add \( D \textbf{ only} \) to node labels containing \( CN \)
  - For \( CN \supseteq D \), add \( \neg D \textbf{ only} \) to node labels containing \( \neg CN \)
  - Can expand definitions lazily
    - Only add definitions \textbf{ after} other local (propositional) expansion
    - Only add definitions one step at a time
Absorption II

Absorb into existing primitive definitions, e.g.

Use lazy expansion technique with primitive definitions

Performance improvements often too large to measure

At least four orders of magnitude

Implementation – p. 9/14
Absorption II

Transform GCIs into primitive definitions, e.g.
Absorption II

Transform GCIs into primitive definitions, e.g.

- $CN \cap C \sqsubseteq D \rightarrow CN \sqsubseteq D \sqcup \neg C$
Absorption II

Transform GCIs into primitive definitions, e.g.

- $CN \cap C \subseteq D \rightarrow CN \subseteq D \cup \neg C$
- $CN \cup C \supseteq D \rightarrow CN \supseteq D \cap \neg C$
Absorption II

- Transform GCls into primitive definitions, e.g.
  - $\text{CN} \cap C \subseteq D \rightarrow \text{CN} \subseteq D \cup \neg C$
  - $\text{CN} \cup C \supset D \rightarrow \text{CN} \supset D \cap \neg C$

- Absorb into existing primitive definitions, e.g.
Absorption II

- Transform GCIs into primitive definitions, e.g.
  - $CN \cap C \subseteq D \rightarrow CN \subseteq D \uplus \neg C$
  - $CN \uplus C \supseteq D \rightarrow CN \supseteq D \cap \neg C$

- Absorb into existing primitive definitions, e.g.
  - $CN \subseteq A, CN \subseteq D \uplus \neg C \rightarrow CN \subseteq A \cap (D \uplus \neg C)$
Absorption II

Transform GCIs into primitive definitions, e.g.
- \( \text{CN} \cap C \subseteq D \rightarrow \text{CN} \subseteq D \cup \neg C \)
- \( \text{CN} \cup C \supseteq D \rightarrow \text{CN} \supseteq D \cap \neg C \)

Absorb into existing primitive definitions, e.g.
- \( \text{CN} \subseteq A, \text{CN} \subseteq D \cup \neg C' \rightarrow \text{CN} \subseteq A \cap (D \cup \neg C') \)
- \( \text{CN} \supseteq A, \text{CN} \supseteq D \cap \neg C' \rightarrow \text{CN} \supseteq A \cup (D \cap \neg C') \)
Absorption II

- Transform GCIs into primitive definitions, e.g.
  - $\text{CN} \cap C \subseteq D \rightarrow \text{CN} \subseteq D \cup \neg C$
  - $\text{CN} \cup C \supseteq D \rightarrow \text{CN} \supseteq D \cap \neg C$

- Absorb into existing primitive definitions, e.g.
  - $\text{CN} \subseteq A$, $\text{CN} \subseteq D \cup \neg C \rightarrow \text{CN} \subseteq A \cap (D \cup \neg C)$
  - $\text{CN} \supseteq A$, $\text{CN} \supseteq D \cap \neg C \rightarrow \text{CN} \supseteq A \cup (D \cap \neg C)$

- Use lazy expansion technique with primitive definitions
Absorption II

- Transform GCIs into primitive definitions, e.g.
  - $CN \cap C \subseteq D \rightarrow CN \subseteq D \cup \neg C$
  - $CN \cup C \supseteq D \rightarrow CN \supseteq D \cap \neg C$

- Absorb into existing primitive definitions, e.g.
  - $CN \subseteq A, CN \subseteq D \cup \neg C \rightarrow CN \subseteq A \cap (D \cup \neg C)$
  - $CN \supseteq A, CN \supseteq D \cap \neg C \rightarrow CN \supseteq A \cup (D \cap \neg C)$

- Use lazy expansion technique with primitive definitions
  - Disjunctions only added to “relevant” node labels
Absorption II

Transform GCIs into primitive definitions, e.g.
- \( CN \sqcap C \subseteq D \rightarrow CN \subseteq D \sqcup \neg C \)
- \( CN \sqcup C \supseteq D \rightarrow CN \supseteq D \cap \neg C \)

Absorb into existing primitive definitions, e.g.
- \( CN \subseteq A, \ CN \subseteq D \sqcup \neg C \rightarrow CN \subseteq A \cap (D \sqcup \neg C) \)
- \( CN \supseteq A, \ CN \supseteq D \cap \neg C \rightarrow CN \supseteq A \cup (D \cap \neg C) \)

Use lazy expansion technique with primitive definitions
- Disjunctions only added to “relevant” node labels

Performance improvements often too large to measure
Absorption II

- Transform GCIs into primitive definitions, e.g.
  - \( CN \cap C \subseteq D \rightarrow CN \subseteq D \cup \neg C \)
  - \( CN \cup C \supseteq D \rightarrow CN \supseteq D \cap \neg C \)

- Absorb into existing primitive definitions, e.g.
  - \( CN \subseteq A, \ CN \subseteq D \cup \neg C \rightarrow CN \subseteq A \cap (D \cup \neg C) \)
  - \( CN \supseteq A, \ CN \supseteq D \cap \neg C \rightarrow CN \supseteq A \cup (D \cap \neg C) \)

- Use lazy expansion technique with primitive definitions
  - Disjunctions only added to “relevant” node labels

- Performance improvements often too large to measure
  - At least **four orders of magnitude** with **GALEN KB**
Algorithmic Optimisations

Useful techniques include

- Avoiding redundancy in search branches
  - Davis-Putnam style semantic branching search
  - Syntactic branching with no-good list
- Dependency directed backtracking
  - Backjumping
  - Dynamic backtracking
- Caching
  - Cache partial models
  - Cache satisfiability status (of labels)
- Heuristic ordering of propositional and modal expansion
  - Min/maximise constrainedness (e.g., MOMS)
  - Maximise backtracking (e.g., oldest first)
Algorithmic Optimisations

Useful techniques include

☞ Avoiding redundancy in search branches
Algorithmic Optimisations

Useful techniques include

☞ Avoiding redundancy in search branches
  - Davis-Putnam style semantic branching search

☞ Dependency directed backtracking
  - Backjumping
  - Dynamic backtracking

☞ Caching
  - Cache partial models
  - Cache satisfiability status (of labels)

☞ Heuristic ordering of propositional and modal expansion
  - Min/maximise constrainedness (e.g., MOMS)
  - Maximise backtracking (e.g., oldest first)
Algorithmic Optimisations

Useful techniques include

☞ Avoiding redundancy in search branches
  - Davis-Putnam style semantic branching search
  - Syntactic branching with no-good list

☞ Dependency directed backtracking
  - Backjumping
  - Dynamic backtracking

☞ Caching
  - Cache partial models
  - Cache satisfiability status (of labels)

☞ Heuristic ordering of propositional and modal expansion
  - Min/maximise constrainedness (e.g., MOMS)
  - Maximise backtracking (e.g., oldest first)
Algorithmic Optimisations

Useful techniques include

☞ Avoiding redundancy in search branches
  - Davis-Putnam style semantic branching search
  - Syntactic branching with no-good list

☞ Dependency directed backtracking

☞ Caching
  - Cache partial models
  - Cache satisfiability status (of labels)

☞ Heuristic ordering of propositional and modal expansion
  - Min/maximise constrainedness (e.g., MOMS)
  - Maximise backtracking (e.g., oldest first)
Algorithmic Optimisations

Useful techniques include

☞ Avoiding redundancy in search branches
  ● Davis-Putnam style semantic branching search
  ● Syntactic branching with no-good list

☞ Dependency directed backtracking
  ● Backjumping
Algorithmic Optimisations

Useful techniques include

☞ Avoiding redundancy in search branches
  - Davis-Putnam style semantic branching search
  - Syntactic branching with no-good list

☞ Dependency directed backtracking
  - Backjumping
  - Dynamic backtracking

Implementation – p. 10/14
Algorithmic Optimisations

Useful techniques include

☞ Avoiding redundancy in search branches
  - Davis-Putnam style semantic branching search
  - Syntactic branching with no-good list

☞ Dependency directed backtracking
  - Backjumping
  - Dynamic backtracking

☞ Caching

Implementation – p. 10/14
Algorithmic Optimisations

Useful techniques include

- Avoiding redundancy in search branches
  - Davis-Putnam style semantic branching search
  - Syntactic branching with no-good list

- Dependency directed backtracking
  - Backjumping
  - Dynamic backtracking

- Caching
  - Cache partial models
Algorithmic Optimisations

Useful techniques include

☞ Avoiding redundancy in search branches
  • Davis-Putnam style semantic branching search
  • Syntactic branching with no-good list

☞ Dependency directed backtracking
  • Backjumping
  • Dynamic backtracking

☞ Caching
  • Cache partial models
  • Cache satisfiability status (of labels)
Algorithmic Optimisations

Useful techniques include

☞ Avoiding redundancy in search branches
  - Davis-Putnam style semantic branching search
  - Syntactic branching with no-good list

☞ Dependency directed backtracking
  - Backjumping
  - Dynamic backtracking

☞ Caching
  - Cache partial models
  - Cache satisfiability status (of labels)

☞ Heuristic ordering of propositional and modal expansion
Algorithmic Optimisations

Useful techniques include

☞ Avoiding redundancy in search branches
  • Davis-Putnam style semantic branching search
  • Syntactic branching with no-good list

☞ Dependency directed backtracking
  • Backjumping
  • Dynamic backtracking

☞ Caching
  • Cache partial models
  • Cache satisfiability status (of labels)

☞ Heuristic ordering of propositional and modal expansion
  • Min/maximise constrainedness (e.g., MOMS)
Algorithmic Optimisations

Useful techniques include

☞ Avoiding redundancy in search branches
  - Davis-Putnam style semantic branching search
  - Syntactic branching with no-good list

☞ Dependency directed backtracking
  - Backjumping
  - Dynamic backtracking

☞ Caching
  - Cache partial models
  - Cache satisfiability status (of labels)

☞ Heuristic ordering of propositional and modal expansion
  - Min/maximise constrainedness (e.g., MOMS)
  - Maximise backtracking (e.g., oldest first)
Dependency Directed Backtracking

- Allows rapid recovery from bad branching choices
- Most commonly used technique is backjumping
- Tag concepts introduced at branch points (e.g., when expanding disjunctions)
- Expansion rules combine and propagate tags
- On discovering a clash, identify most recently introduced concepts involved
- Jump back to relevant branch points without exploring alternative branches
- Effect is to prune away part of the search space
- Performance improvements with GALEN KB again too large to measure
Dependency Directed Backtracking

- Allows rapid recovery from bad branching choices
Dependency Directed Backtracking

- Allows rapid recovery from bad branching choices
- Most commonly used technique is backjumping
Dependency Directed Backtracking

- Allows rapid recovery from bad branching choices
- Most commonly used technique is backjumping
  - Tag concepts introduced at branch points (e.g., when expanding disjunctions)
Dependency Directed Backtracking

- Allows rapid recovery from bad branching choices
- Most commonly used technique is **backjumping**
  - Tag concepts introduced at branch points (e.g., when expanding disjunctions)
  - Expansion rules combine and propagate tags
Dependency Directed Backtracking

- Allows rapid recovery from bad branching choices
- Most commonly used technique is **backjumping**
  - Tag concepts introduced at branch points (e.g., when expanding disjunctions)
  - Expansion rules combine and propagate tags
  - On discovering a clash, identify most recently introduced concepts involved
Dependency Directed Backtracking

- Allows rapid recovery from bad branching choices
- Most commonly used technique is **backjumping**
  - Tag concepts introduced at branch points (e.g., when expanding disjunctions)
  - Expansion rules combine and propagate tags
  - On discovering a clash, identify most recently introduced concepts involved
  - Jump back to relevant branch points **without exploring** alternative branches
Dependency Directed Backtracking

- Allows rapid recovery from bad branching choices
- Most commonly used technique is **backjumping**
  - Tag concepts introduced at branch points (e.g., when expanding disjunctions)
  - Expansion rules combine and propagate tags
  - On discovering a clash, identify most recently introduced concepts involved
  - Jump back to relevant branch points **without exploring** alternative branches
  - Effect is to prune away part of the search space
Dependency Directed Backtracking

- Allows rapid recovery from bad branching choices
- Most commonly used technique is **backjumping**
  - Tag concepts introduced at branch points (e.g., when expanding disjunctions)
  - Expansion rules combine and propagate tags
  - On discovering a clash, identify most recently introduced concepts involved
  - Jump back to relevant branch points **without exploring** alternative branches
  - Effect is to prune away part of the search space
  - Performance improvements with GALEN KB again **too large to measure**
Backjumping

E.g., if $\exists R. \neg A \cap \forall R. (A \cap B) \cap (C_1 \cup D_1) \cap \ldots \cap (C_n \cup D_n) \subseteq \mathcal{L}(x)$
Backjumping

E.g., if $\exists R. \neg A \quad \forall R. (A \cap B) \cap (C_1 \cup D_1) \cap \ldots \cap (C_n \cup D_n) \subseteq \mathcal{L}(x)$

$x$
Backjumping

E.g., if \( \exists R. \neg A \cap \forall R. (A \cap B) \cap (C_1 \cup D_1) \cap \ldots \cap (C_n \cup D_n) \subseteq \mathcal{L}(x) \)
Backjumping

E.g., if \( \exists R. \neg A \land \forall R. (A \land B) \land (C_1 \lor D_1) \land \ldots \land (C_n \lor D_n) \subseteq \mathcal{L}(x) \)
Backjumping

E.g., if $\exists R. \neg A \cap \forall R. (A \cap B) \cap (C_1 \cup D_1) \cap \ldots \cap (C_n \cup D_n) \subseteq \mathcal{L}(x)$
Backjumping

E.g., if $\exists R. \neg A \land \forall R. (A \land B) \land (C_1 \lor D_1) \land \ldots \land (C_n \lor D_n) \subseteq \mathcal{L}(x)$

\[
\mathcal{L}(x) \cup \{C_1\} \quad \mathcal{L}(x) \cup \{C_{n-1}\} \quad \mathcal{L}(x) \cup \{C_n\} \quad \mathcal{L}(y) = \{(A \land B), \neg A, A, B\}
\]

Clash
Backjumping

E.g., if $\exists R. \neg A \land \forall R. (A \land B) \land (C_1 \lor D_1) \land \ldots \land (C_n \lor D_n) \subseteq \mathcal{L}(x)$
Backjumping

E.g., if \( \exists R. \neg A \cap \forall R.(A \cap B) \cap (C_1 \cup D_1) \cap \ldots \cap (C_n \cup D_n) \subseteq \mathcal{L}(x) \)

\[
\begin{align*}
\mathcal{L}(x) & \cup \{C_1\} \\
\mathcal{L}(x) & \cup \{\neg C_1, D_1\} \\
\mathcal{L}(x) & \cup \{C_2\} \\
\mathcal{L}(x) & \cup \{\neg C_2, D_2\} \\
\mathcal{L}(x) & \cup \{\neg C_n, D_n\} \\
\mathcal{L}(x) & \cup \{\neg C_n, D_n\}
\end{align*}
\]

\[
\begin{align*}
\mathcal{L}(y) & = \{(A \cap B), \neg A, A, B\} \\
\mathcal{L}(y) & = \{(A \cap B), \neg A, A, B\}
\end{align*}
\]

Clash  Clash  Clash  …  Clash
Backjumping

E.g., if $\exists R. \neg A \cap \forall R. (A \cap B) \cap (C_1 \cup D_1) \cap \ldots \cap (C_n \cup D_n) \subseteq \mathcal{L}(x)$
Backjumping

E.g., if $\exists R. \neg A \land \forall R. (A \land B) \land (C_1 \cup D_1) \land \ldots \land (C_n \cup D_n) \subseteq \mathcal{L}(x)$
Caching

Identical node labels often recur during expansion. Avoid re-solving problems by caching satisfiability status.

When \( L(x) \) initialised, look in cache. Use result, or add status once it has been computed. Can use sub/super set caching to deal with similar labels. Care required when used with blocking or inverse roles. Significant performance gains with some kinds of problem.

Cache (partial) models of concepts. Use to detect “obvious” non-subsumption:

\[
C_u : D \text{ satisfiable if models of } C \text{ and } D \text{ can be merged}
\]

If not, continue with standard subsumption test. Can use same technique in sub-problems.

Implementation – p. 13/14
Caching

* Cache the satisfiability status of a node label

Identical node labels often recur during expansion. Avoid re-solving problems by caching satisfiability status. When \( L(x) \) initialised, look in cache. Use result, or add status once it has been computed. Can use sub/super set caching to deal with similar labels. Care required when used with blocking or inverse roles. Significant performance gains with some kinds of problem.

* Cache (partial) models of concepts

Use to detect "obvious" non-subsumption. If \( C_u : D \) is satisfiable, \( C_u : D \) satisfiable if models of \( C \) and \( D \) can be merged. If not, continue with standard subsumption test. Can use same technique in sub-problems.
Caching

- Cache the satisfiability status of a node label
  - Identical node labels often recur during expansion
Caching

- Cache the satisfiability status of a node label
  - Identical node labels often recur during expansion
  - Avoid re-solving problems by caching satisfiability status

Implementation – p. 13/14
Caching

- Cache the satisfiability status of a node label
  - Identical node labels often recur during expansion
  - Avoid re-solving problems by caching satisfiability status
    - When $\mathcal{L}(x)$ initialised, look in cache

- Can use sub/super set caching to deal with similar labels
  - Care required when used with blocking or inverse roles
  - Significant performance gains with some kinds of problem

- Cache (partial) models of concepts
  - Use to detect "obvious" non-subsumption

If not, continue with standard subsumption test

Can use same technique in sub-problems

Implementation – p. 13/14
Caching

Cache the satisfiability status of a node label
- Identical node labels often recur during expansion
- Avoid re-solving problems by caching satisfiability status
  ➔ When $\mathcal{L}(x)$ initialised, look in cache
  ➔ Use result, or add status once it has been computed
Caching

- Cache the satisfiability status of a node label
  - Identical node labels often recur during expansion
  - Avoid re-solving problems by caching satisfiability status
    - When $\mathcal{L}(x)$ initialised, look in cache
    - Use result, or add status once it has been computed
  - Can use sub/super set caching to deal with similar labels
Caching

- Cache the satisfiability status of a node label
  - Identical node labels often recur during expansion
  - Avoid re-solving problems by caching satisfiability status
    - When $\mathcal{L}(x)$ initialised, look in cache
    - Use result, or add status once it has been computed
  - Can use sub/super set caching to deal with similar labels
  - Care required when used with blocking or inverse roles
Caching

Cache the satisfiability status of a node label

- Identical node labels often recur during expansion
- Avoid re-solving problems by caching satisfiability status
  - When $L(x)$ initialised, look in cache
  - Use result, or add status once it has been computed
- Can use sub/super set caching to deal with similar labels
- Care required when used with blocking or inverse roles
- Significant performance gains with some kinds of problem
Cache the satisfiability status of a node label

- Identical node labels often recur during expansion
- Avoid re-solving problems by caching satisfiability status
  - When $\mathcal{L}(x)$ initialised, look in cache
  - Use result, or add status once it has been computed
- Can use sub/super set caching to deal with similar labels
- Care required when used with blocking or inverse roles
- Significant performance gains with some kinds of problem

Cache (partial) models of concepts
Caching

- Cache the satisfiability status of a node label
  - Identical node labels often recur during expansion
  - Avoid re-solving problems by caching satisfiability status
    ➞ When $\mathcal{L}(x)$ initialised, look in cache
    ➞ Use result, or add status once it has been computed
  - Can use sub/super set caching to deal with similar labels
  - Care required when used with blocking or inverse roles
  - Significant performance gains with some kinds of problem

- Cache (partial) models of concepts
  - Use to detect “obvious” non-subsumption
Caching

- Cache the satisfiability status of a node label
  - Identical node labels often recur during expansion
  - Avoid re-solving problems by caching satisfiability status
    ➞ When $L(x)$ initialised, look in cache
    ➞ Use result, or add status once it has been computed
  - Can use sub/super set caching to deal with similar labels
  - Care required when used with blocking or inverse roles
  - Significant performance gains with some kinds of problem

- Cache (partial) models of concepts
  - Use to detect “obvious” non-subsumption
  - $C \not\subseteq D$ if $C \cap \neg D$ is satisfiable
Caching

- Cache the satisfiability status of a node label
  - Identical node labels often recur during expansion
  - Avoid re-solving problems by caching satisfiability status
    ➔ When \( \mathcal{L}(x) \) initialised, look in cache
    ➔ Use result, or add status once it has been computed
  - Can use sub/super set caching to deal with similar labels
  - Care required when used with blocking or inverse roles
  - Significant performance gains with some kinds of problem

- Cache (partial) models of concepts
  - Use to detect “obvious” non-subsumption
  - \( C \nsubseteq D \) if \( C \cap \neg D \) is satisfiable
  - \( C \cap \neg D \) satisfiable if models of \( C \) and \( \neg D \) can be merged
Caching

- Cache the satisfiability status of a node label
  - Identical node labels often recur during expansion
  - Avoid re-solving problems by caching satisfiability status
    ➔ When $\mathcal{L}(x)$ initialised, look in cache
    ➔ Use result, or add status once it has been computed
  - Can use sub/super set caching to deal with similar labels
  - Care required when used with blocking or inverse roles
  - Significant performance gains with some kinds of problem

- Cache (partial) models of concepts
  - Use to detect “obvious” non-subsumption
  - $C \not\subseteq D$ if $C \cap \neg D$ is satisfiable
  - $C \cap \neg D$ satisfiable if models of $C$ and $\neg D$ can be merged
  - If not, continue with standard subsumption test
Caching

- Cache the satisfiability status of a node label
  - Identical node labels often recur during expansion
  - Avoid re-solving problems by caching satisfiability status
  - When $\mathcal{L}(x)$ initialised, look in cache
  - Use result, or add status once it has been computed
  - Can use sub/super set caching to deal with similar labels
  - Care required when used with blocking or inverse roles
  - Significant performance gains with some kinds of problem

- Cache (partial) models of concepts
  - Use to detect “obvious” non-subsumption
  - $C \nsubseteq D$ if $C \cap \neg D$ is satisfiable
  - $C \cap \neg D$ satisfiable if models of $C$ and $\neg D$ can be merged
  - If not, continue with standard subsumption test
  - Can use same technique in sub-problems
Summary

• Naive implementation results in effective non-termination
• Problem is caused by non-deterministic expansion (search)
• GCIs lead to huge search space
• Solution (partial) is careful choice of logic/algorithm
  - Avoid encodings
  - Highly optimised implementation
• Most important optimisations are:
  - Absorption
  - Dependency directed backtracking (backjumping)
  - Caching
• Performance improvements can be very large
  - E.g., more than four orders of magnitude
Summary

Naive implementation results in effective non-termination
Summary

- Naive implementation results in effective non-termination
- Problem is caused by non-deterministic expansion (search)

Most important optimisations are:
- Absorption
- Dependency directed backtracking (backjumping)
- Caching

Performance improvements can be very large
E.g., more than four orders of magnitude
Summary

- Naive implementation results in effective non-termination
- Problem is caused by non-deterministic expansion (search)
  - GCIs lead to huge search space

Most important optimisations are:
- Absorption
- Dependency directed backtracking (backjumping)
- Caching

Performance improvements can be very large
E.g., more than four orders of magnitude
Summary

- Naive implementation results in effective non-termination
- Problem is caused by non-deterministic expansion (*search*)
  - GCIs lead to huge search space
- Solution (partial) is
Summary

- Naive implementation results in effective non-termination
- Problem is caused by non-deterministic expansion (*search*)
  - GCIs lead to huge search space
- Solution (partial) is
  - Careful choice of logic/algorithm

Performance improvements can be very large
  E.g., more than four orders of magnitude
Summary

- Naive implementation results in effective non-termination
- Problem is caused by non-deterministic expansion (*search*)
  - GCI s lead to huge search space
- Solution (partial) is
  - Careful choice of logic/algorithm
  - Avoid encodings

Performance improvements can be very large
  - E.g., more than four orders of magnitude
Summary

- Naive implementation results in effective non-termination
- Problem is caused by non-deterministic expansion (search)
  - GCIs lead to huge search space
- Solution (partial) is
  - Careful choice of logic/algorithm
  - Avoid encodings
  - Highly optimised implementation
Summary

- Naive implementation results in effective non-termination
- Problem is caused by non-deterministic expansion (search)
  - GCIs lead to huge search space
- Solution (partial) is
  - Careful choice of logic/algorithm
  - Avoid encodings
  - Highly optimised implementation
- Most important optimisations are
Summary

- Naive implementation results in effective non-termination
- Problem is caused by non-deterministic expansion (search)
  - GCIs lead to huge search space
- Solution (partial) is
  - Careful choice of logic/algorithm
  - Avoid encodings
  - Highly optimised implementation
- Most important optimisations are
  - Absorption
Summary

- Naive implementation results in effective non-termination
- Problem is caused by non-deterministic expansion (search)
  - GCIs lead to huge search space
- Solution (partial) is
  - Careful choice of logic/algorithm
  - Avoid encodings
  - Highly optimised implementation
- Most important optimisations are
  - Absorption
  - Dependency directed backtracking (backjumping)
Summary

- Naive implementation results in effective non-termination
- Problem is caused by non-deterministic expansion (search)
  - GCIs lead to huge search space
- Solution (partial) is
  - Careful choice of logic/algorithm
  - Avoid encodings
  - Highly optimised implementation
- Most important optimisations are
  - Absorption
  - Dependency directed backtracking (backjumping)
  - Caching
Summary

- Naive implementation results in effective non-termination
- Problem is caused by non-deterministic expansion (search)
  - GCIs lead to huge search space
- Solution (partial) is
  - Careful choice of logic/algorithm
  - Avoid encodings
  - Highly optimised implementation
- Most important optimisations are
  - Absorption
  - Dependency directed backtracking (backjumping)
  - Caching
- Performance improvements can be very large
Summary

- Naive implementation results in effective non-termination
- Problem is caused by non-deterministic expansion (search)
  - GCIs lead to huge search space
- Solution (partial) is
  - Careful choice of logic/algorithm
  - Avoid encodings
  - Highly optimised implementation
- Most important optimisations are
  - Absorption
  - Dependency directed backtracking (backjumping)
  - Caching
- Performance improvements can be very large
  - E.g., more than four orders of magnitude