Implementing DL Systems

Naive Implementations

Problems include:

Naive Implementations

Problems include:

- ☞ Space usage

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

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:

- 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**

Careful Choice of Algorithm

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 $R \in \mathbf{R}_+$

Careful Choice of Algorithm

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

Careful Choice of Algorithm

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

Careful Choice of Algorithm

- Transitive roles instead of transitive closure
  - Deterministic expansion of $\exists R.C$, even when $R \in \mathbf{R}_+$
  - (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 $R \in \mathbf{R}_+$
  - (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 $R \in \mathbf{R}_+$
  - (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, . . .

Careful Choice of Algorithm

- Transitive roles instead of transitive closure
  - Deterministic expansion of $\exists R.C$, even when $R \in \mathbf{R}_+$
  - (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., $($domain $R.C) \equiv \exists R.\top \sqsubseteq C$

Careful Choice of Algorithm

- Transitive roles instead of transitive closure
  - Deterministic expansion of $\exists R.C$, even when $R \in \mathbf{R}_+$
  - (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., $($domain $R.C) \equiv \exists R.\top \sqsubseteq C$
  - (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 $R \in \mathbf{R}_+$
  - (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., $(\text{domain } R.C) \equiv \exists R.\top \sqsubseteq C$
  - (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

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

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

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

Optimising Subsumption Testing

Optimisation techniques broadly fall into 2 categories

- ☞ Pre-processing optimisations

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

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

Pre-processing Optimisations

Useful techniques include

- ☞ Normalisation and simplification of concepts

Pre-processing Optimisations

Useful techniques include

- ☞ Normalisation and simplification of concepts
  - Refinement of technique first used in $\mathcal{KRIS}$ system

Pre-processing Optimisations

Useful techniques include

- ☞ Normalisation and simplification of concepts
  - Refinement of technique first used in $\mathcal{KRIS}$ 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 $\mathcal{KRIS}$ 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 $\mathcal{KRIS}$ 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 $\mathcal{KRIS}$ 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 $\mathcal{KRIS}$ 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 $\mathcal{KRIS}$ 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 $\mathcal{KRIS}$ 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 $\mathcal{KRIS}$ 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 $\mathcal{KRIS}$ 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

Normalisation and Simplification

☞ Normalise concepts to standard form, e.g.:

Normalisation and Simplification

☞ Normalise concepts to standard form, e.g.:

- $\exists R.C \longrightarrow \neg\forall R.\neg C$

Normalisation and Simplification

☞ Normalise concepts to standard form, e.g.:

- $\exists R.C \longrightarrow \neg\forall R.\neg C$
- $C \sqcup D \longrightarrow \neg(\neg C \sqcap \neg D)$

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.:

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.:
  - $(D \sqcap C) \sqcap (A \sqcap D) \longrightarrow A \sqcap C \sqcap D$

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.:
  - $(D \sqcap C) \sqcap (A \sqcap D) \longrightarrow A \sqcap C \sqcap D$
  - $\forall R.\top \longrightarrow \top$

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.:
  - $(D \sqcap C) \sqcap (A \sqcap D) \longrightarrow A \sqcap C \sqcap D$
  - $\forall R.\top \longrightarrow \top$
  - $\ldots \sqcap C \sqcap \ldots \sqcap \neg C \sqcap \ldots \longrightarrow \bot$

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.:
  - $\exists R.C \longrightarrow \neg\forall R.\neg C$
  - $C \sqcup D \longrightarrow \neg(\neg C \sqcap \neg D)$
- 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
  - Use names/pointers to refer to complex concepts

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

- 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.:
  - $\exists R.C \longrightarrow \neg\forall R.\neg C$
  - $C \sqcup D \longrightarrow \neg(\neg C \sqcap \neg D)$
- 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
  - 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.:
  - $\exists R.C \longrightarrow \neg\forall R.\neg C$
  - $C \sqcup D \longrightarrow \neg(\neg C \sqcap \neg D)$
- 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
  - Use names/pointers to refer to complex concepts
  - Only add structure as required by progress of algorithm
  - Detect clashes between lexically equivalent concepts

$\{\text{HappyFather}, \neg\text{HappyFather}\} \longrightarrow$ **clash**

$\{\forall\text{has-child}.(\text{Doctor} \sqcup \text{Lawyer}), \exists\text{has-child}.(\neg\text{Doctor} \sqcap \neg\text{Lawyer})\} \longrightarrow$ **search**

Absorption I

Absorption I

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

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

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

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}

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

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

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

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 $\sqsupseteq D$, 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 $C \sqsubseteq D$ 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 $\sqsupseteq D$, 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 $\sqsupseteq D$, 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 $C \sqsubseteq D$ 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 $\sqsupseteq D$, add $\neg D$ **only** to node labels containing $\neg$CN
  - Can expand definitions lazily
    - → Only add definitions **after** other local (propositional) expansion
    - → Only add definitions one step at a time

Absorption II

Absorption II

- Transform GCIs into primitive definitions, e.g.

Absorption II

- Transform GCIs into primitive definitions, e.g.
  - $\mathsf{CN} \sqcap C \sqsubseteq D \longrightarrow \mathsf{CN} \sqsubseteq D \sqcup \neg C$

Absorption II

- Transform GCIs into primitive definitions, e.g.
  - $\mathsf{CN} \sqcap C \sqsubseteq D \longrightarrow \mathsf{CN} \sqsubseteq D \sqcup \neg C$
  - $\mathsf{CN} \sqcup C \sqsupseteq D \longrightarrow \mathsf{CN} \sqsupseteq D \sqcap \neg C$

Absorption II

- Transform GCIs into primitive definitions, e.g.
  - $\mathsf{CN} \sqcap C \sqsubseteq D \longrightarrow \mathsf{CN} \sqsubseteq D \sqcup \neg C$
  - $\mathsf{CN} \sqcup C \sqsupseteq D \longrightarrow \mathsf{CN} \sqsupseteq D \sqcap \neg C$
- Absorb into existing primitive definitions, e.g.

Absorption II

- Transform GCIs into primitive definitions, e.g.
  - $\mathsf{CN} \sqcap C \sqsubseteq D \longrightarrow \mathsf{CN} \sqsubseteq D \sqcup \neg C$
  - $\mathsf{CN} \sqcup C \sqsupseteq D \longrightarrow \mathsf{CN} \sqsupseteq D \sqcap \neg C$
- Absorb into existing primitive definitions, e.g.
  - $\mathsf{CN} \sqsubseteq A$, $\mathsf{CN} \sqsubseteq D \sqcup \neg C \longrightarrow \mathsf{CN} \sqsubseteq A \sqcap (D \sqcup \neg C)$

Absorption II

- Transform GCIs into primitive definitions, e.g.
  - $\mathsf{CN} \sqcap C \sqsubseteq D \longrightarrow \mathsf{CN} \sqsubseteq D \sqcup \neg C$
  - $\mathsf{CN} \sqcup C \sqsupseteq D \longrightarrow \mathsf{CN} \sqsupseteq D \sqcap \neg C$
- Absorb into existing primitive definitions, e.g.
  - $\mathsf{CN} \sqsubseteq A$, $\mathsf{CN} \sqsubseteq D \sqcup \neg C \longrightarrow \mathsf{CN} \sqsubseteq A \sqcap (D \sqcup \neg C)$
  - $\mathsf{CN} \sqsupseteq A$, $\mathsf{CN} \sqsupseteq D \sqcap \neg C \longrightarrow \mathsf{CN} \sqsupseteq A \sqcup (D \sqcap \neg C)$

Absorption II

- ☞ Transform GCIs into primitive definitions, e.g.
  - $\mathsf{CN} \sqcap C \sqsubseteq D \longrightarrow \mathsf{CN} \sqsubseteq D \sqcup \neg C$
  - $\mathsf{CN} \sqcup C \sqsupseteq D \longrightarrow \mathsf{CN} \sqsupseteq D \sqcap \neg C$
- ☞ Absorb into existing primitive definitions, e.g.
  - $\mathsf{CN} \sqsubseteq A$, $\mathsf{CN} \sqsubseteq D \sqcup \neg C \longrightarrow \mathsf{CN} \sqsubseteq A \sqcap (D \sqcup \neg C)$
  - $\mathsf{CN} \sqsupseteq A$, $\mathsf{CN} \sqsupseteq D \sqcap \neg C \longrightarrow \mathsf{CN} \sqsupseteq A \sqcup (D \sqcap \neg C)$
- ☞ Use lazy expansion technique with primitive definitions

Absorption II

- Transform GCIs into primitive definitions, e.g.
  - CN $\sqcap C \sqsubseteq D \longrightarrow$ CN $\sqsubseteq D \sqcup \neg C$
  - CN $\sqcup C \sqsupseteq D \longrightarrow$ CN $\sqsupseteq D \sqcap \neg C$
- Absorb into existing primitive definitions, e.g.
  - CN $\sqsubseteq A$, CN $\sqsubseteq D \sqcup \neg C \longrightarrow$ CN $\sqsubseteq A \sqcap (D \sqcup \neg C)$
  - CN $\sqsupseteq A$, CN $\sqsupseteq D \sqcap \neg C \longrightarrow$ CN $\sqsupseteq A \sqcup (D \sqcap \neg C)$
- Use lazy expansion technique with primitive definitions
  - Disjunctions only added to “relevant” node labels

Absorption II

- Transform GCIs into primitive definitions, e.g.
  - CN $\sqcap C \sqsubseteq D \longrightarrow$ CN $\sqsubseteq D \sqcup \neg C$
  - CN $\sqcup C \sqsupseteq D \longrightarrow$ CN $\sqsupseteq D \sqcap \neg C$
- Absorb into existing primitive definitions, e.g.
  - CN $\sqsubseteq A$, CN $\sqsubseteq D \sqcup \neg C \longrightarrow$ CN $\sqsubseteq A \sqcap (D \sqcup \neg C)$
  - CN $\sqsupseteq A$, CN $\sqsupseteq D \sqcap \neg C \longrightarrow$ CN $\sqsupseteq A \sqcup (D \sqcap \neg C)$
- 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 $\sqcap C \sqsubseteq D \longrightarrow$ CN $\sqsubseteq D \sqcup \neg C$
  - CN $\sqcup C \sqsupseteq D \longrightarrow$ CN $\sqsupseteq D \sqcap \neg C$
- Absorb into existing primitive definitions, e.g.
  - CN $\sqsubseteq A$, CN $\sqsubseteq D \sqcup \neg C \longrightarrow$ CN $\sqsubseteq A \sqcap (D \sqcup \neg C)$
  - CN $\sqsupseteq A$, CN $\sqsupseteq D \sqcap \neg C \longrightarrow$ CN $\sqsupseteq A \sqcup (D \sqcap \neg C)$
- 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

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

Algorithmic Optimisations

Useful techniques include

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

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

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

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

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

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

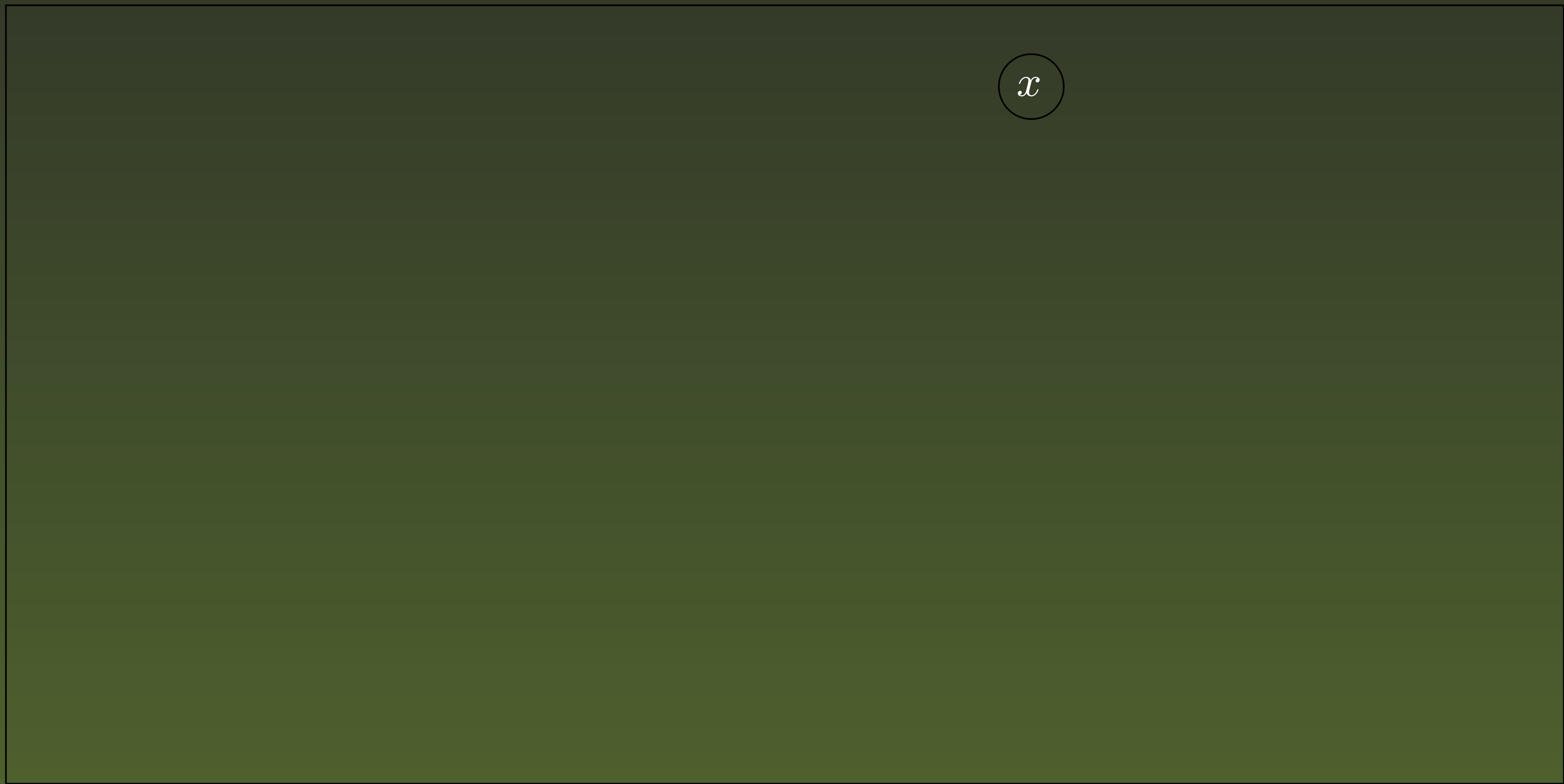
- Allows rapid recovery from bad branching choices
- Most commonly used technique is **backjumping**
  - Tag concepts introduced at branch points (e.g., when expanding disjunctions)
  - Expansion rules combine and propagate tags
  - On discovering a clash, identify most recently introduced concepts involved
  - 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 $\exists R.\neg A \sqcap \forall R.(A \sqcap B) \sqcap (C_1 \sqcup D_1) \sqcap \ldots \sqcap (C_n \sqcup D_n) \subseteq \mathcal{L}(x)$

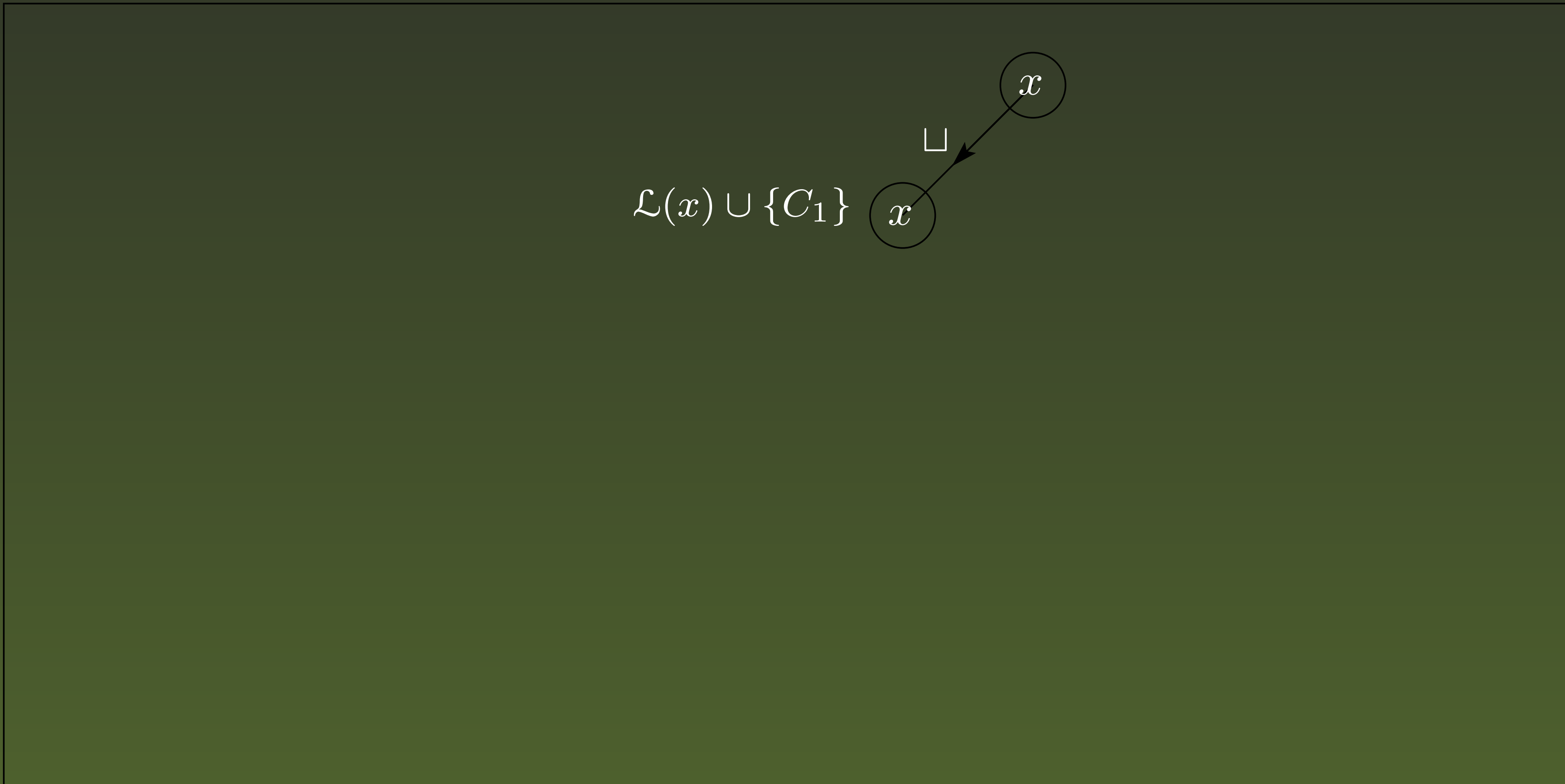
Backjumping

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



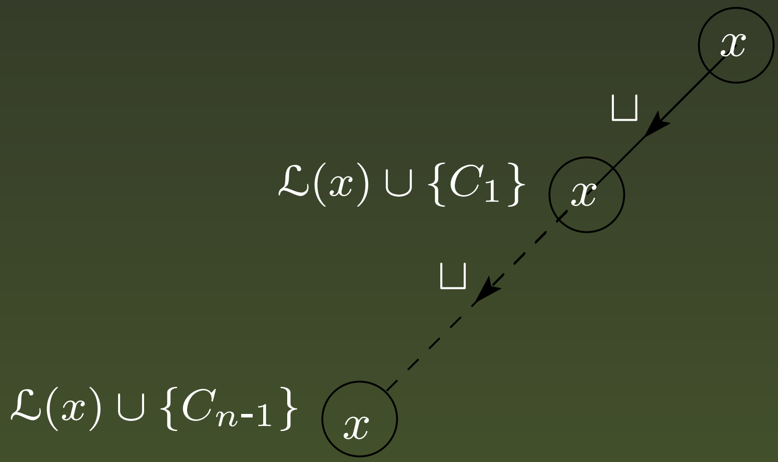
Backjumping

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



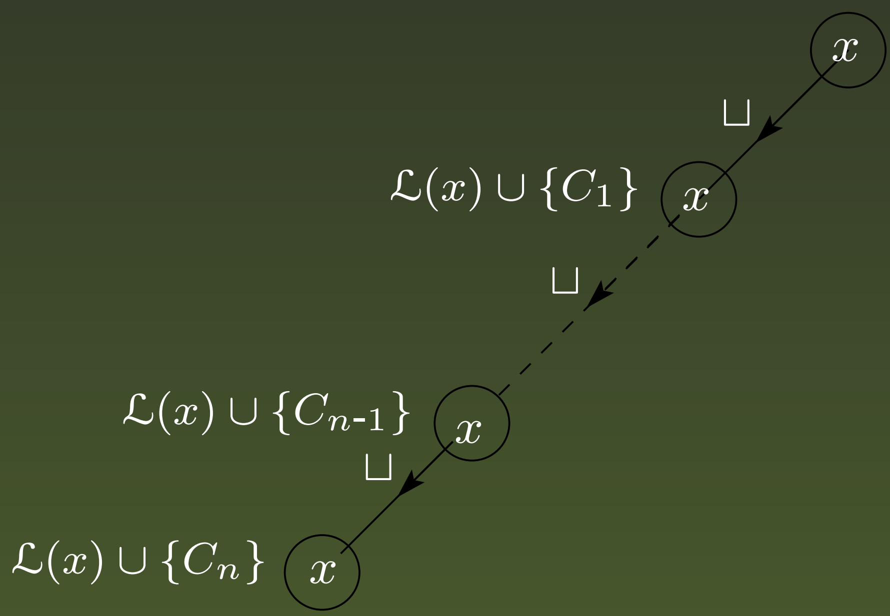
Backjumping

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



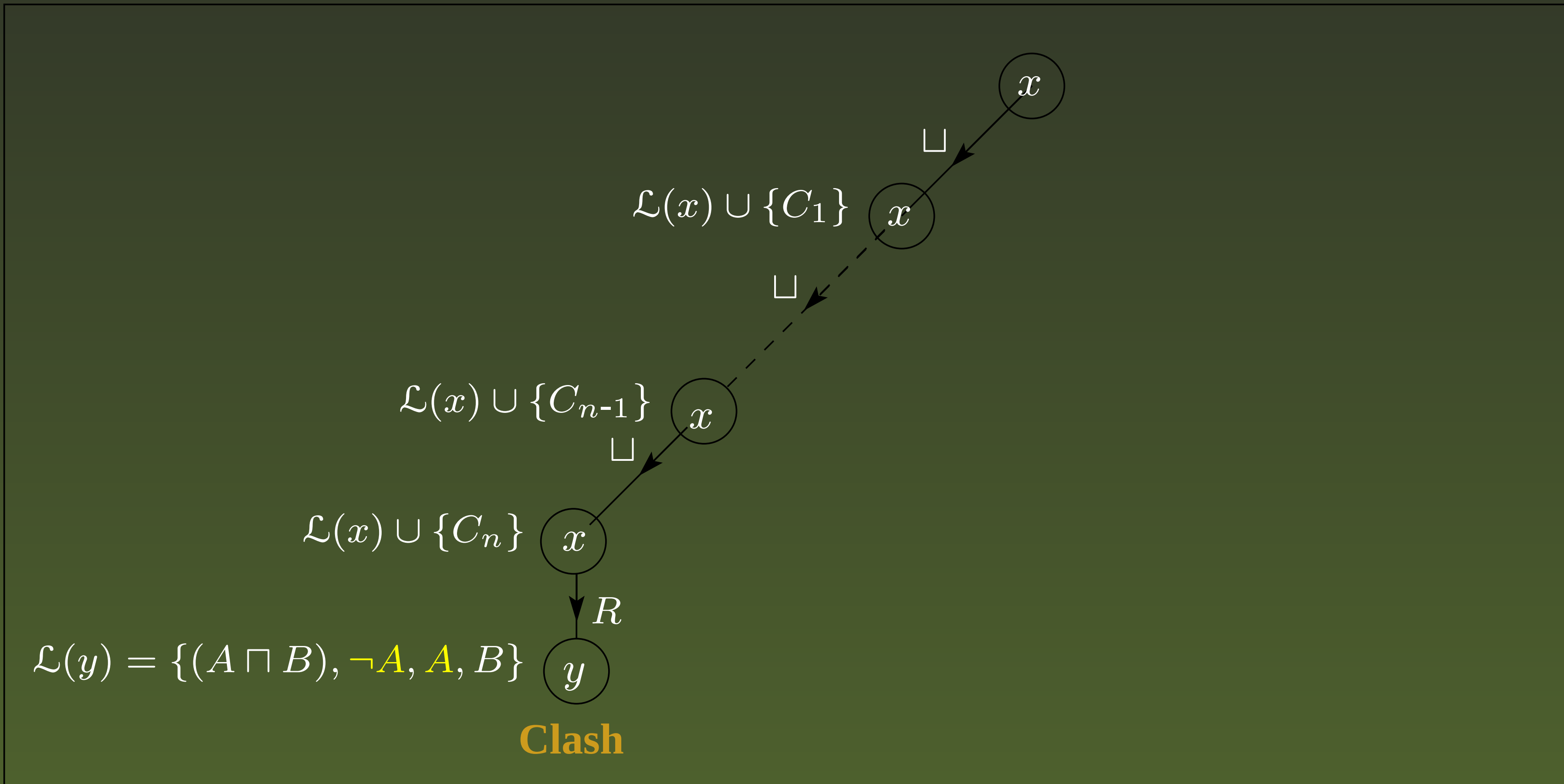
Backjumping

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



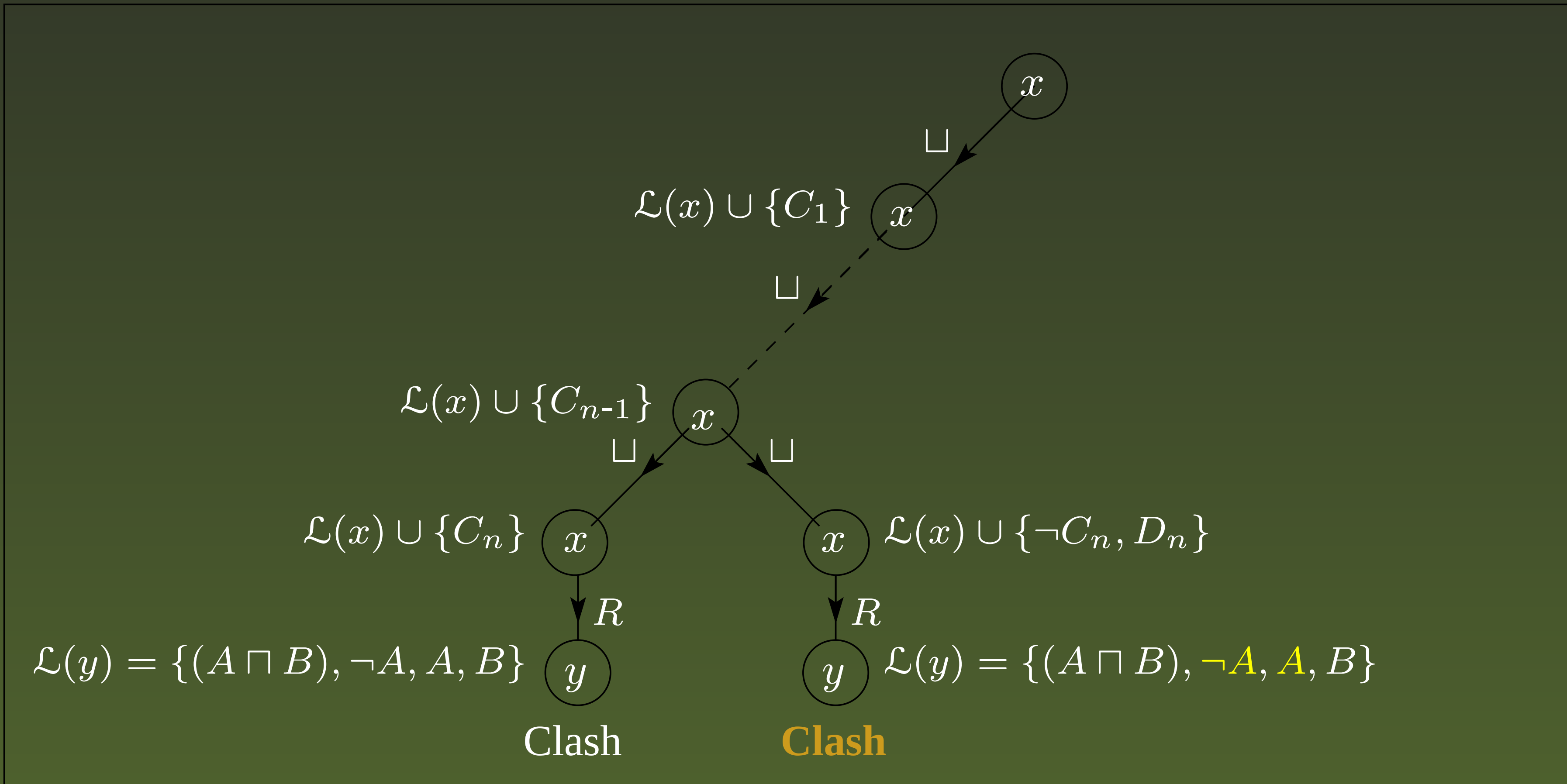
Backjumping

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



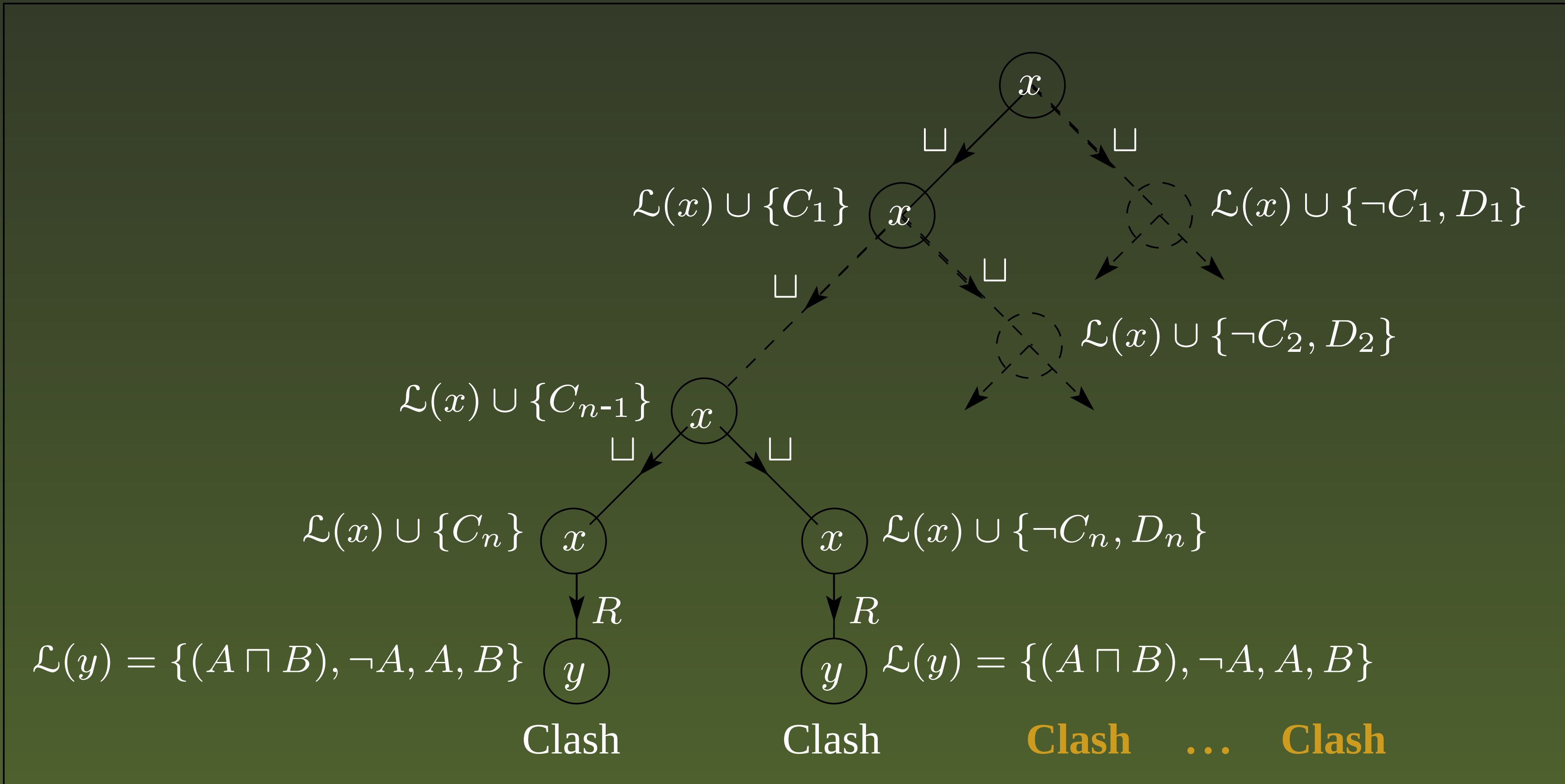
Backjumping

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



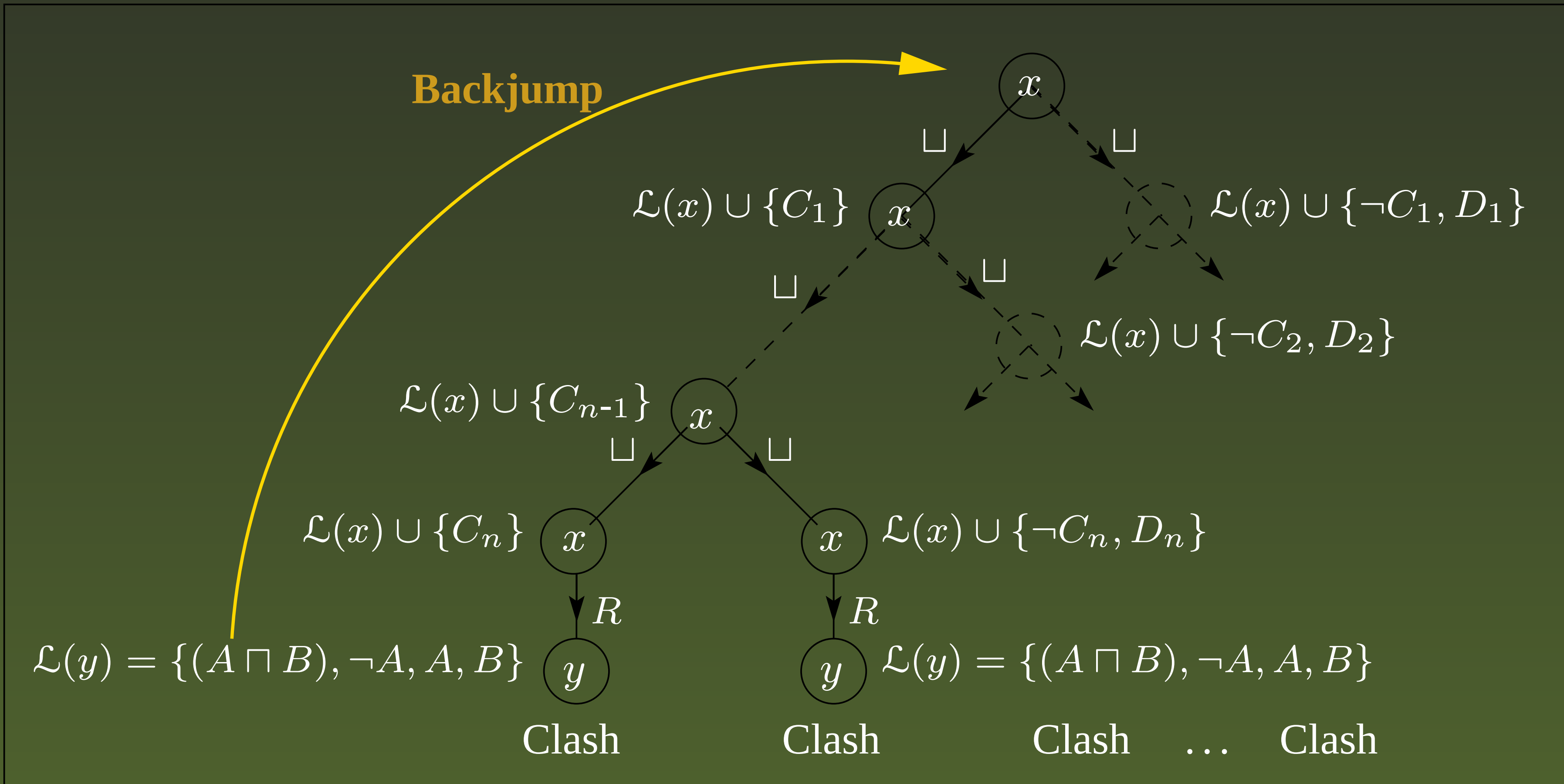
Backjumping

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



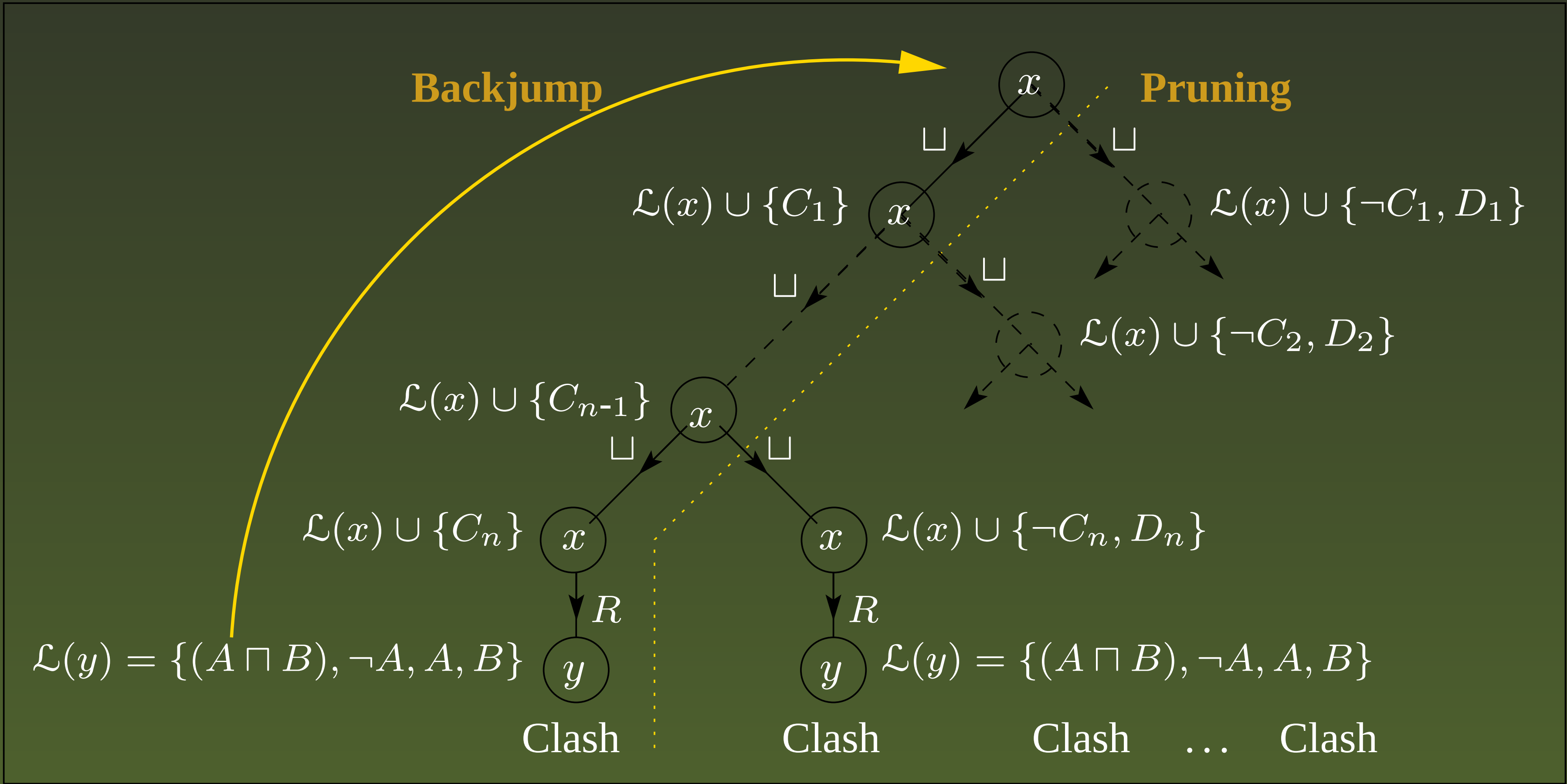
Backjumping

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



Backjumping

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



Caching

Caching

☞ Cache the satisfiability status of a node label

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

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

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 $\mathcal{L}(x)$ 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

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

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 $\mathcal{L}(x)$ 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
  - $C \not\sqsubseteq D$ if $C \sqcap \neg D$ 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
  - $C \not\sqsubseteq D$ if $C \sqcap \neg D$ is satisfiable
  - $C \sqcap \neg D$ 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
  - $C \not\sqsubseteq D$ if $C \sqcap \neg D$ is satisfiable
  - $C \sqcap \neg D$ 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
  - $C \not\sqsubseteq D$ if $C \sqcap \neg D$ is satisfiable
  - $C \sqcap \neg D$ 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

Summary

- ☞ Naive implementation results in effective non-termination

Summary

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

Summary

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

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

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

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**

Select Bibliography

F. Baader, E. Franconi, B. Hollunder, B. Nebel, and H.-J. Profitlich. An empirical analysis of optimization techniques for terminological representation systems or: Making KRIS get a move on. In B. Nebel, C. Rich, and W. Swartout, editors, *Proc. of KR'92*, pages 270–281. Morgan Kaufmann, 1992.

F. Giunchiglia and R. Sebastiani. A SAT-based decision procedure for $\mathcal{ALC}$. In *Proc. of KR'96*, pages 304–314. Morgan Kaufmann, 1996.

V. Haarslev and R. Möller. High performance reasoning with very large knowledge bases: A practical case study. In *Proc. of IJCAI 2001* (to appear).

B. Hollunder and W. Nutt. Subsumption algorithms for concept languages. In *Proc. of ECAI'90*, pages 348–353. John Wiley & Sons Ltd., 1990.

Select Bibliography

I. Horrocks. *Optimising Tableaux Decision Procedures for Description Logics*. PhD thesis, University of Manchester, 1997.

I. Horrocks and P. F. Patel-Schneider. Comparing subsumption optimizations. In *Proc. of DL'98*, pages 90–94. CEUR, 1998.

I. Horrocks and P. F. Patel-Schneider. Optimising description logic subsumption. *Journal of Logic and Computation*, 9(3):267–293, 1999.

I. Horrocks and S. Tobies. Reasoning with axioms: Theory and practice. In *Proc. of KR'00* pages 285–296. Morgan Kaufmann, 2000.