Inferring Heap Abstraction Grammars

Alexander Dominik Weinert

RWTH Aachen University

July 30, 2012

Model Checking

Verification by exploration of states

Model Checking on Unbounded Structures

Model Checking

Verification by exploration of states

Input: int i_1, int i_2
  while $i_1 \neq 0$ **do**
    $i_1 := i_1 - 1$
    $i_2 := i_2 + 1$
  end while

Finite number of states

Model Checking on Unbounded Structures

Model Checking

Verification by exploration of states

Input: int i_1, int i_2
while $i_1 \neq 0$ **do**
$i_1 := i_1 - 1$
$i_2 := i_2 + 1$
end while

Finite number of states ✓

Model Checking on Unbounded Structures

Model Checking

Verification by exploration of states

Input: int i_1, int i_2
while $i_1 \neq 0$ **do**
$i_1 := i_1 - 1$
$i_2 := i_2 + 1$
end while

Finite number of states ✓

Input: List l
Element $e := l.first()$
while $e \neq null$ **do**
$e := l.next(e)$
end while

Infinite number of states

Model Checking on Unbounded Structures

Model Checking

Verification by exploration of states

Input: int i_1, int i_2
while $i_1 \neq 0$ **do**
$i_1 := i_1 - 1$
$i_2 := i_2 + 1$
end while

Finite number of states ✓

Input: List l
Element $e := l.\mathit{first}()$
while $e \neq \mathit{null}$ **do**
$e := l.\mathit{next}(e)$
end while

Infinite number of states χ

Model Checking on Unbounded Structures

Model Checking

Verification by exploration of states

Input: int i_1, int i_2
while $i_1 \neq 0$ **do**
$i_1 := i_1 - 1$
$i_2 := i_2 + 1$
end while

Finite number of states ✓

Input: List l
Element $e := l.first()$
while $e \neq null$ **do**
$e := l.next(e)$
end while

Infinite number of states χ

Idea [Heinen et al., 2009]

Heap Abstraction Grammars $\Rightarrow$ Finitely many heap states

Alphabets

Traditional Case

An alphabet is a finite set of symbols

Alphabets

Traditional Case

~~An alphabet is a finite set of symbols~~

Alphabets

Traditional Case

~~An alphabet is a finite set of symbols~~

Definition

An alphabet is a triple: $\Sigma := (N, T, rk)$,

Alphabets

Traditional Case

~~An alphabet is a finite set of symbols~~

Definition

An alphabet is a triple: $\Sigma := (N, T, rk)$,

where

- N – nonterminal symbols
- T – terminal symbols
- rk: $N \cup T \to \mathbb{N}$ – ranking function

Alphabets

Traditional Case

~~An alphabet is a finite set of symbols~~

Definition

An alphabet is a triple: $\Sigma := (N, T, rk)$,

where

- N – nonterminal symbols
- T – terminal symbols
- rk: $N \cup T \to \mathbb{N}$ – ranking function

and $N \cap T = \emptyset$

Heap Configurations

Definition

A heap configuration over an alphabet Σ and a finite set of symbols Γ is a tuple

$$G := (\qquad\qquad\qquad\qquad\qquad)$$

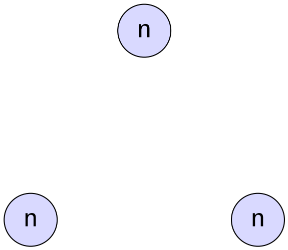
Heap Configurations

Definition

A heap configuration over an alphabet Σ and a finite set of symbols Γ is a tuple

$$G := (V \quad , labV \qquad\qquad)$$

- V – nodes
- labV: $V \to \Gamma$ – node labels



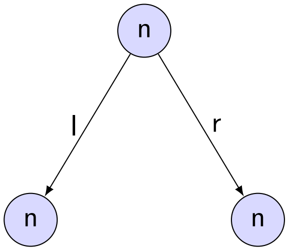
Heap Configurations

Definition

A heap configuration over an alphabet Σ and a finite set of symbols Γ is a tuple

$$G := (V, E, labV, labE, att \qquad)$$

- V – nodes
- E – edges
- labV: $V \to \Gamma$ – node labels
- labE: $E \to N \cup T$ – edge labels
- att: $E \to V^*$ – attachment



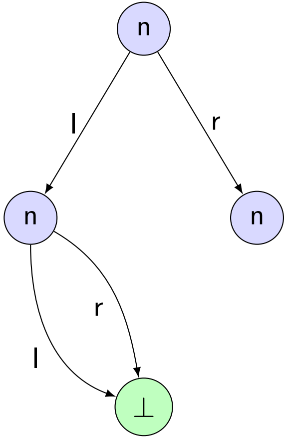
Heap Configurations

Definition

A heap configuration over an alphabet Σ and a finite set of symbols Γ is a tuple

$$G := (V, E, labV, labE, att \quad , \bot)$$

- V – nodes
- E – edges
- labV: $V \to \Gamma$ – node labels
- labE: $E \to N \cup T$ – edge labels
- att: $E \to V^*$ – attachment
- $\bot \in V$ – null node



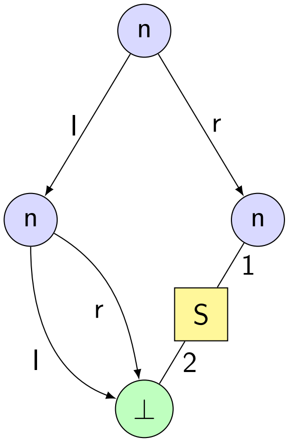
Heap Configurations

Definition

A heap configuration over an alphabet Σ and a finite set of symbols Γ is a tuple

$$G := (V, E, labV, labE, att \qquad, \bot)$$

- V – nodes
- E – edges
- labV: $V \to \Gamma$ – node labels
- labE: $E \to N \cup T$ – edge labels
- att: $E \to V^*$ – attachment
- $\bot \in V$ – null node



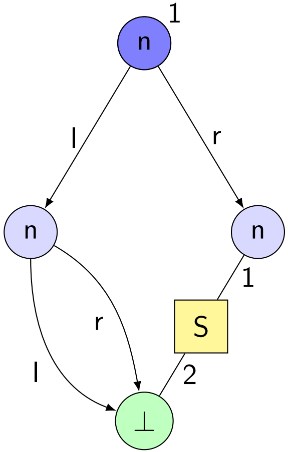
Heap Configurations

Definition

A heap configuration over an alphabet Σ and a finite set of symbols Γ is a tuple

$$G := (V, E, labV, labE, att, ext, \perp)$$

- V – nodes
- E – edges
- labV: $V \to \Gamma$ – node labels
- labE: $E \to N \cup T$ – edge labels
- att: $E \to V^*$ – attachment
- ext $\in V^*$ – external nodes
- $\perp \in V$ – null node



Heap Configurations (cont.)

Definitions

- Rank of an edge := number of nodes attached

Heap Configurations (cont.)

Definitions

- Rank of an edge := number of nodes attached
- Terminal edge $:\Leftrightarrow labE(e) \in T$

Heap Configurations (cont.)

Definitions

- Rank of an edge $:=$ number of nodes attached
- Terminal edge $:\Leftrightarrow labE(e) \in T$
- Terminal heap configuration $:\Leftrightarrow$ all edges are terminal

Heap Configurations (cont.)

Definitions

- Rank of an edge := number of nodes attached
- Terminal edge :$\Leftrightarrow$ $labE(e) \in T$
- Terminal heap configuration :$\Leftrightarrow$ all edges are terminal

Requirements

- Terminal edges are of rank 2

Heap Configurations (cont.)

Definitions

- Rank of an edge := number of nodes attached
- Terminal edge $:\Leftrightarrow labE(e) \in T$
- Terminal heap configuration $:\Leftrightarrow$ all edges are terminal

Requirements

- Terminal edges are of rank 2
- Rank of a label determines rank of edges

Heap Configurations (cont.)

Definitions

- Rank of an edge $:=$ number of nodes attached
- Terminal edge $:\Leftrightarrow labE(e) \in T$
- Terminal heap configuration $:\Leftrightarrow$ all edges are terminal

Requirements

- Terminal edges are of rank 2
- Rank of a label determines rank of edges
- All outgoing pointers of a class are represented

Heap Configurations (cont.)

Definitions

- Rank of an edge := number of nodes attached
- Terminal edge $:\Leftrightarrow labE(e) \in T$
- Terminal heap configuration $:\Leftrightarrow$ all edges are terminal

Requirements

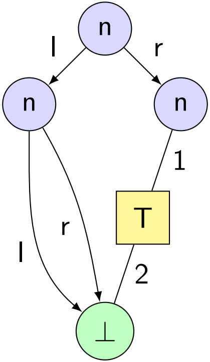
- Terminal edges are of rank 2
- Rank of a label determines rank of edges
- All outgoing pointers of a class are represented
- All pointers are connected according to their type

Hyperedge Substitution

Given:
Assumed:

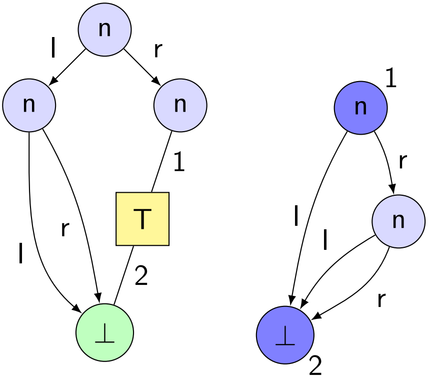
Hyperedge Substitution

Given: heap configurations G, ,edge e
Assumed: $e \in E_G$, $labE(e) \in N$



Hyperedge Substitution

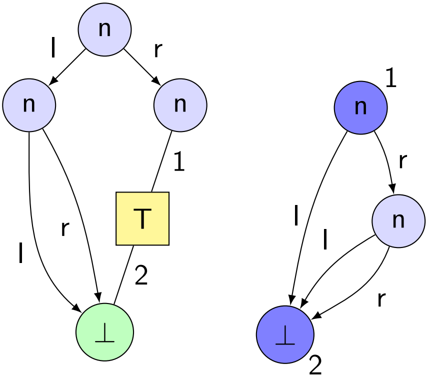
Given: heap configurations G, H ,edge e
Assumed: $e \in E_G$, $labE(e) \in N$



Hyperedge Substitution

Given: heap configurations G, H ,edge e
Assumed: $e \in E_G$, $labE(e) \in N$

Substitute e with H

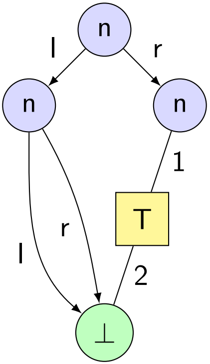


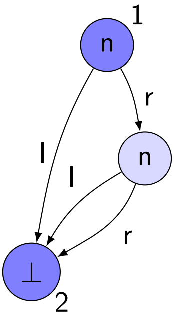
Hyperedge Substitution

Given: heap configurations G, H ,edge e

Assumed: $e \in E_G$, $labE(e) \in N$

Substitute e with H, use external nodes as attachment points.

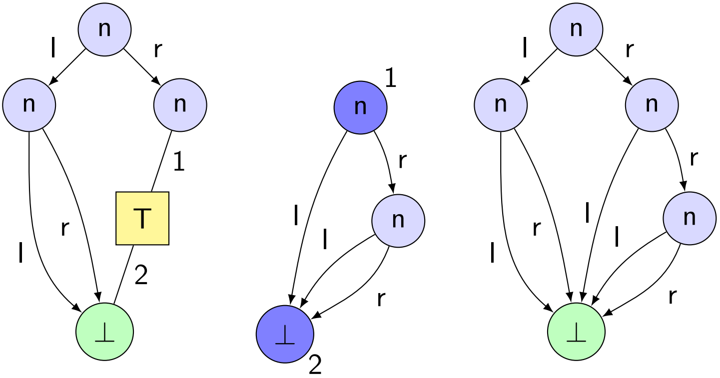




Hyperedge Substitution

Given: heap configurations G, H ,edge e
Assumed: $e \in E_G$, $labE(e) \in N$

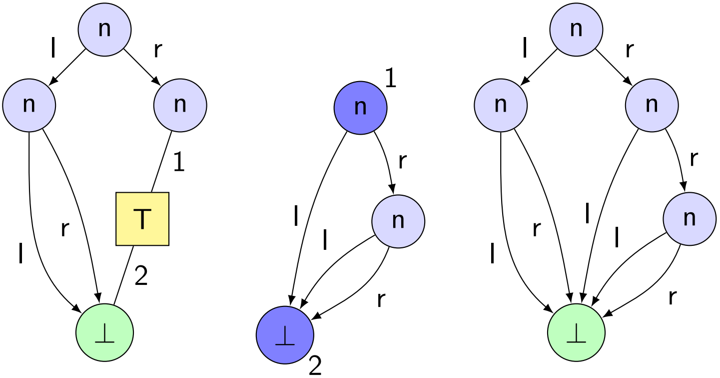
Substitute e with H, use external nodes as attachment points.



Hyperedge Substitution

Given: heap configurations G, H ,edge e
Assumed: $e \in E_G$, $labE(e) \in N$, $rk(e) = |ext_H|$.

Substitute e with H, use external nodes as attachment points.



Data Structure Grammars

Definition

A Data Structure Grammar (DSG) I over an alphabet Σ and a set of symbols Γ is a tuple (N, T, P, S),

Data Structure Grammars

Definition

A Data Structure Grammar (DSG) I over an alphabet Σ and a set of symbols Γ is a tuple (N, T, P, S),

where

- N – nonterminal symbols
- T – terminal symbols
- $P = \{X_1 \to G_1, \ldots, X_n \to G_n\}$ – production rules
- S – axiom

Data Structure Grammars

Definition

A Data Structure Grammar (DSG) I over an alphabet Σ and a set of symbols Γ is a tuple (N, T, P, S),

where

- N – nonterminal symbols
- T – terminal symbols
- $P = \{X_1 \to G_1, \ldots, X_n \to G_n\}$ – production rules
- S – axiom

$L(I) :=$ Set of all terminal configurations that can be derived from S

Heap Abstraction Grammar

Definition

A Heap Abstraction Grammar (HAG) is a Data Structure Grammar (DSG), that

1. Produces only valid heap configurations

Heap Abstraction Grammar

Definition

A Heap Abstraction Grammar (HAG) is a Data Structure Grammar (DSG), that

1. Produces only valid heap configurations
2. Fulfills other restrictions [Heinen et al., tbp]

Heap Abstraction Grammar

Definition

A Heap Abstraction Grammar (HAG) is a Data Structure Grammar (DSG), that

1. Produces only valid heap configurations
2. Fulfills other restrictions [Heinen et al., tbp]

Any DSG that fulfils the first condition, but not the second one can be transformed into a HAG [Jansen, 2010].

Problem

Given: Set of Heap Configurations L

Problem

Given: Set of Heap Configurations L

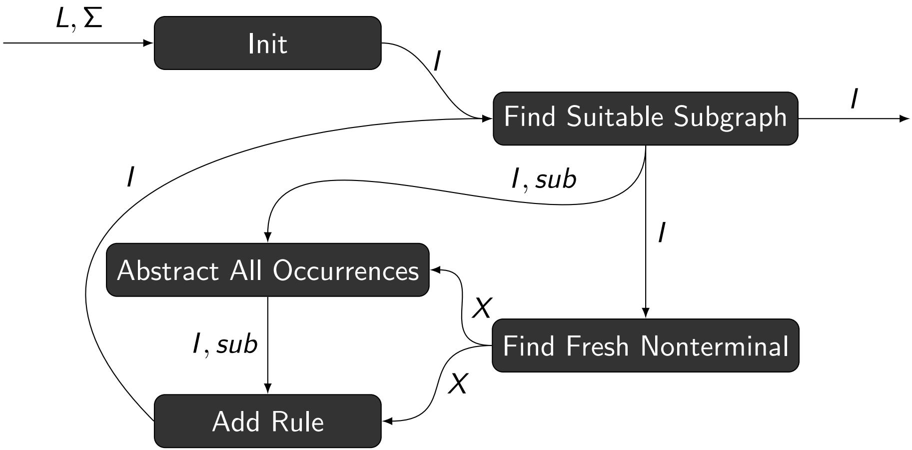
Goal: Heap Abstraction Grammar I with $L(I) \supseteq L$

Overview

Overview

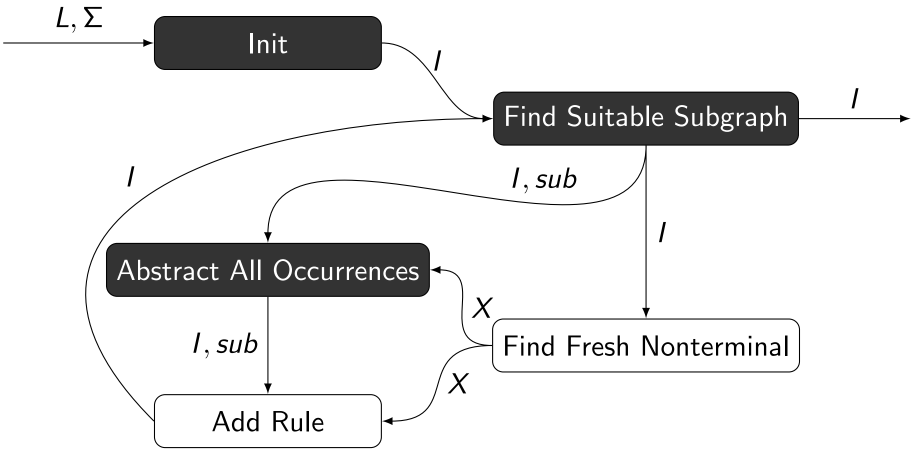
First: $L(I) = L$

Overview



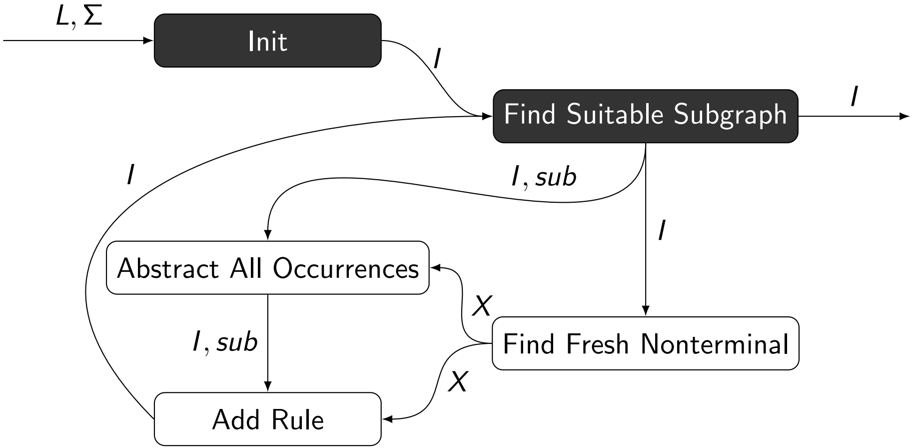
L, Σ
Init
I
Find Suitable Subgraph
I
I
I, sub
I
Abstract All Occurrences
X
Find Fresh Nonterminal
I, sub
X
Add Rule

Overview



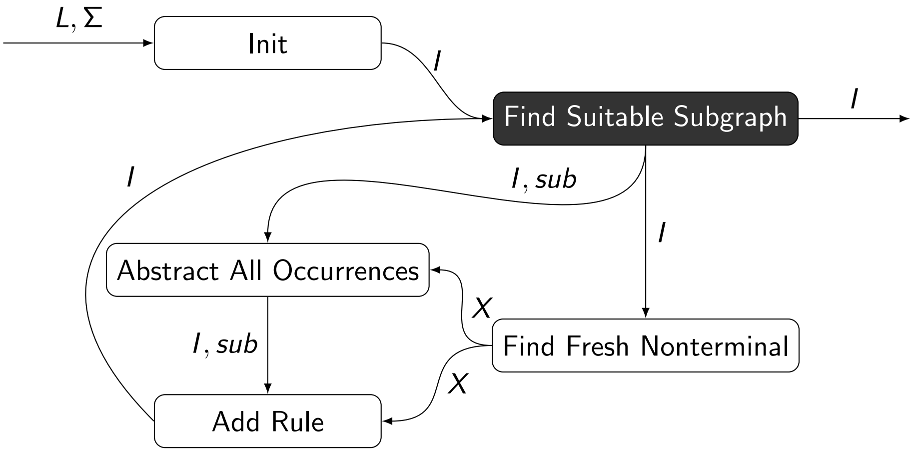
L, Σ
Init
I
Find Suitable Subgraph
I
I
I, sub
I
Abstract All Occurrences
X
Find Fresh Nonterminal
I, sub
X
Add Rule

Overview



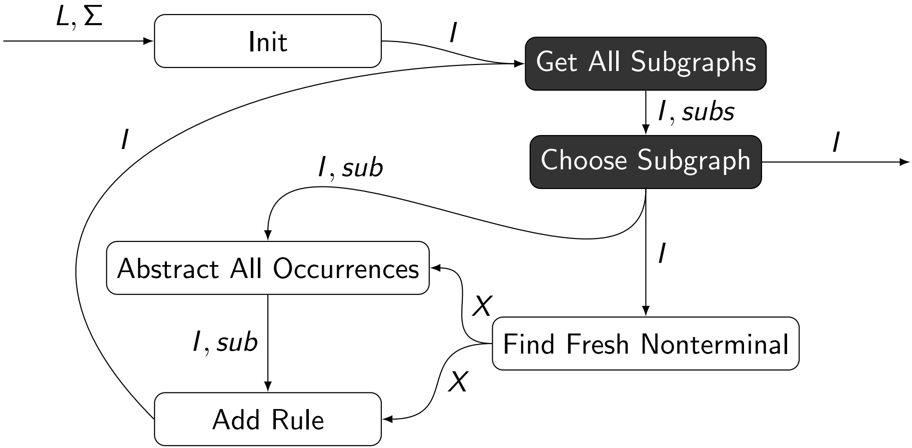
L, Σ
Init
I
Find Suitable Subgraph
I
I
I, sub
I
Abstract All Occurrences
X
Find Fresh Nonterminal
I, sub
X
Add Rule

Overview



L, Σ
Init
I
Find Suitable Subgraph
I
I
I, sub
I
Abstract All Occurrences
X
Find Fresh Nonterminal
I, sub
X
Add Rule

Overview



L, Σ
Init
I
Get All Subgraphs
$I, subs$
Choose Subgraph
I
I
I, sub
Abstract All Occurrences
I
X
Find Fresh Nonterminal
I, sub
X
Add Rule

Subgraph Enumeration

Problem: Exponentially many subgraphs

Subgraph Enumeration

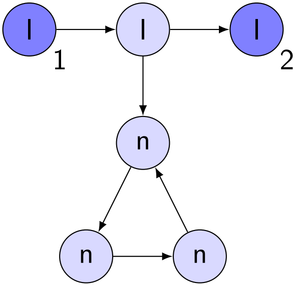
Problem: Exponentially many subgraphs

Solution: Growing subgraphs [Jonyer et al., 2002]

Subgraph Enumeration

Problem: Exponentially many subgraphs

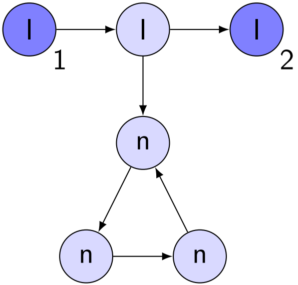
Solution: Growing subgraphs [Jonyer et al., 2002]

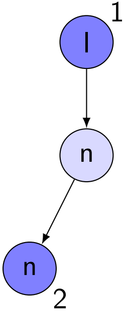


Subgraph Enumeration

Problem: Exponentially many subgraphs

Solution: Growing subgraphs [Jonyer et al., 2002]

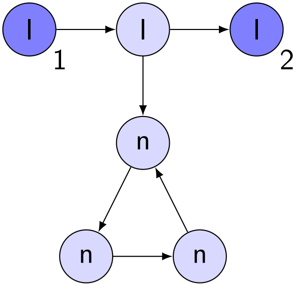


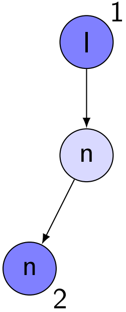


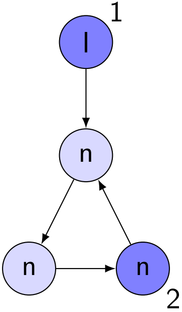
Subgraph Enumeration

Problem: Exponentially many subgraphs

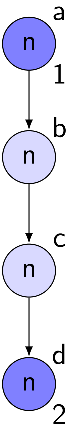
Solution: Growing subgraphs [Jonyer et al., 2002]



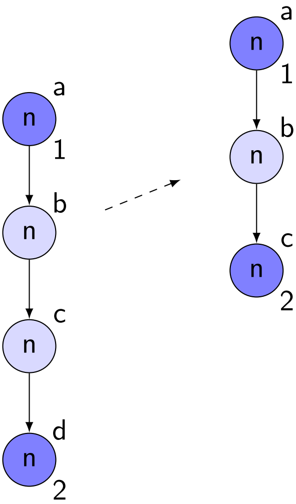




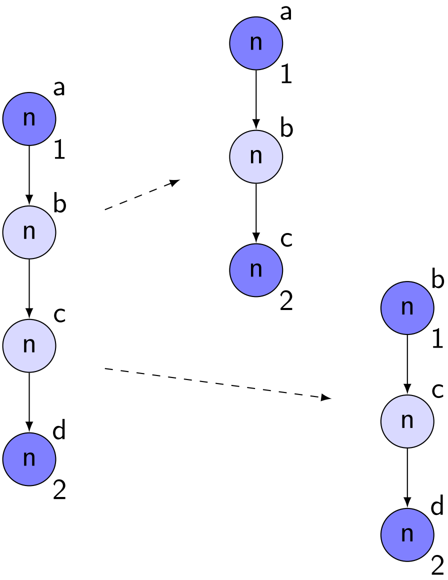
Duplicate and Isomorphic Subgraphs



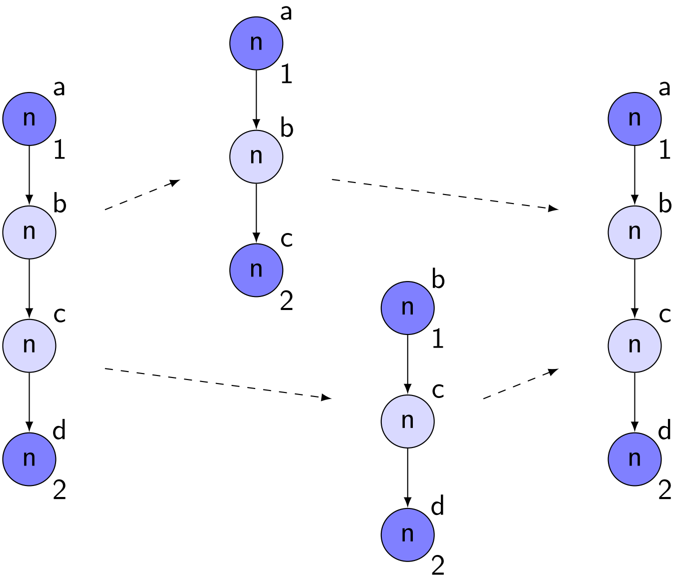
Duplicate and Isomorphic Subgraphs



Duplicate and Isomorphic Subgraphs



Duplicate and Isomorphic Subgraphs



Duplication vs. Isomorphism

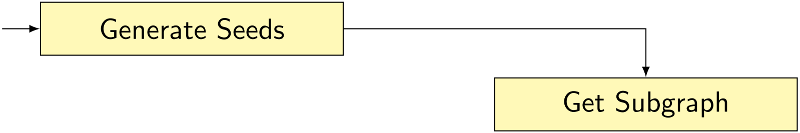
Isomorphic subgraphs $\hat{=}$ Same structure at different points in the graph

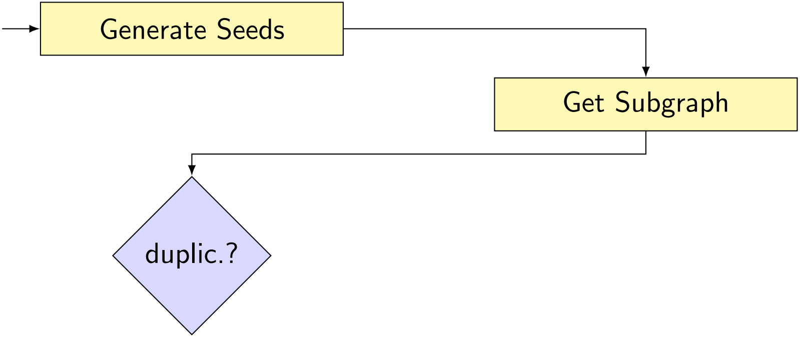
Duplication vs. Isomorphism

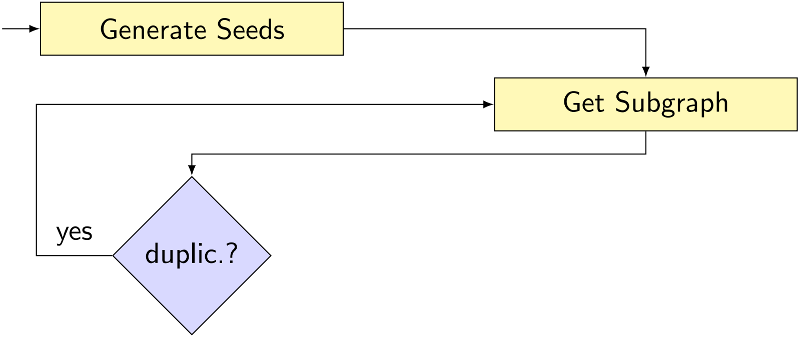
Isomorphic subgraphs $\hat{=}$ Same structure at different points in the graph

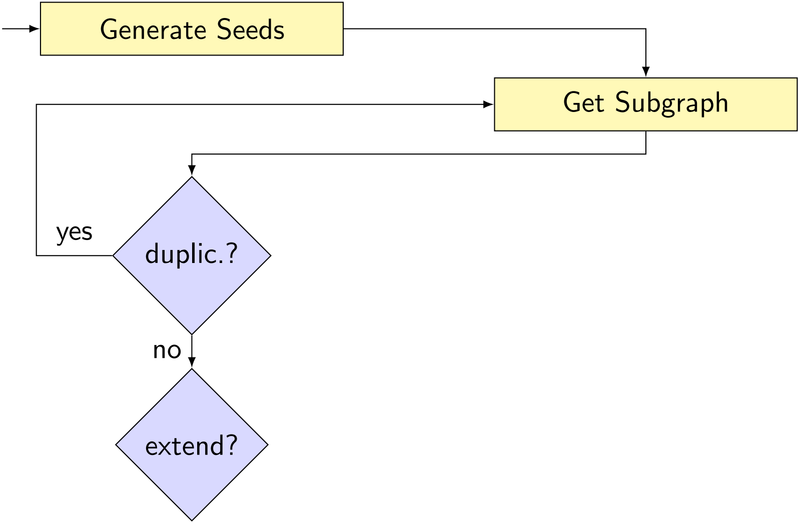
Duplicate subgraphs $\hat{=}$ Same structure at same point in the graph

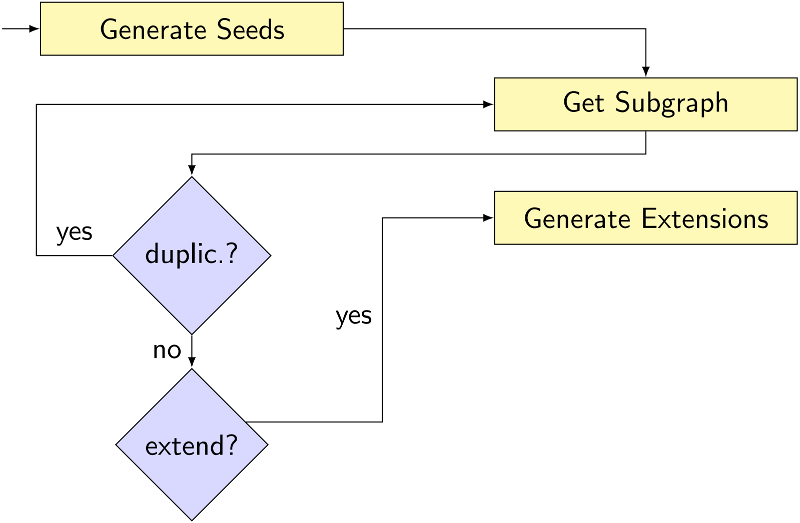
Generate Seeds

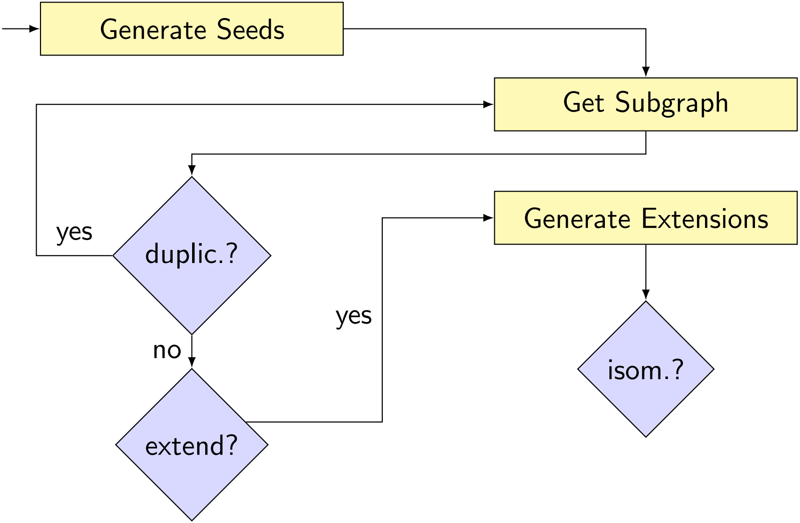

Generate Seeds
Get Subgraph

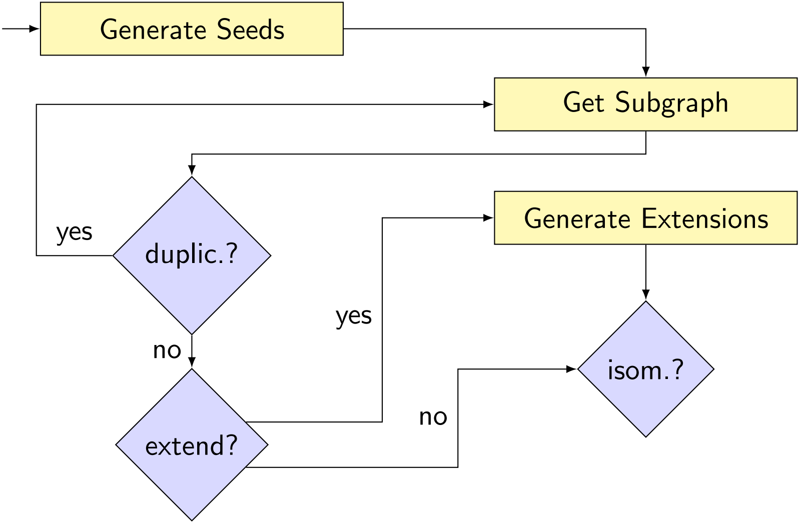

Generate Seeds
Get Subgraph
duplic.?

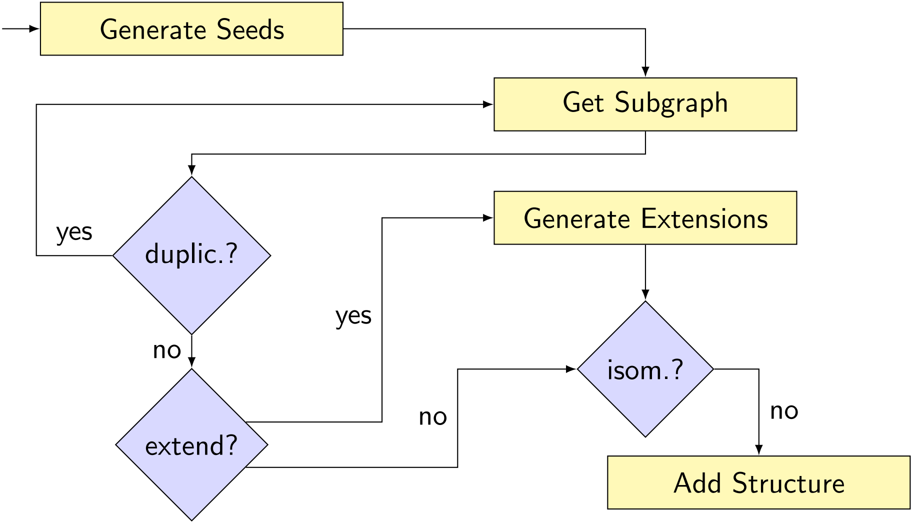

Generate Seeds
Get Subgraph
yes
duplic.?

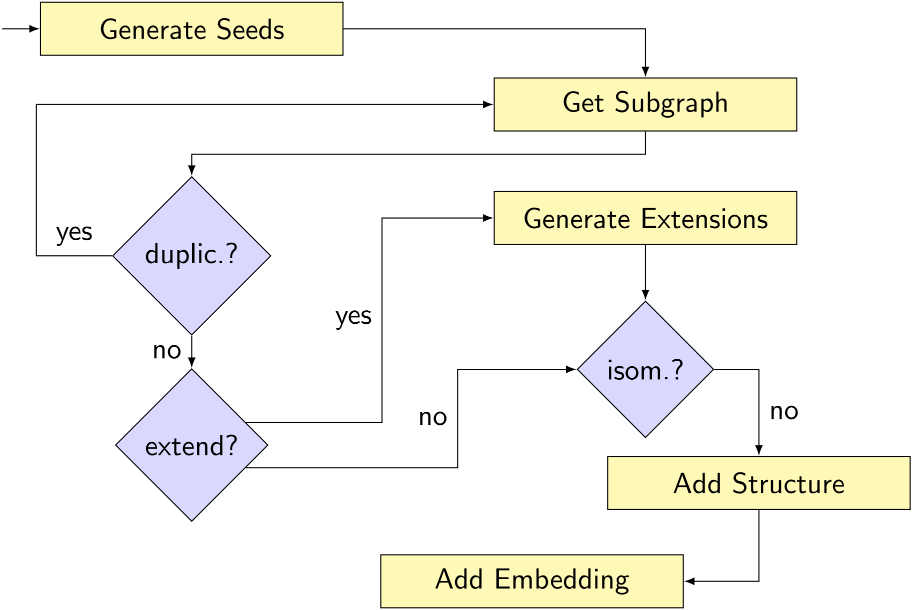

Generate Seeds
Get Subgraph
yes
duplic.?
no
extend?

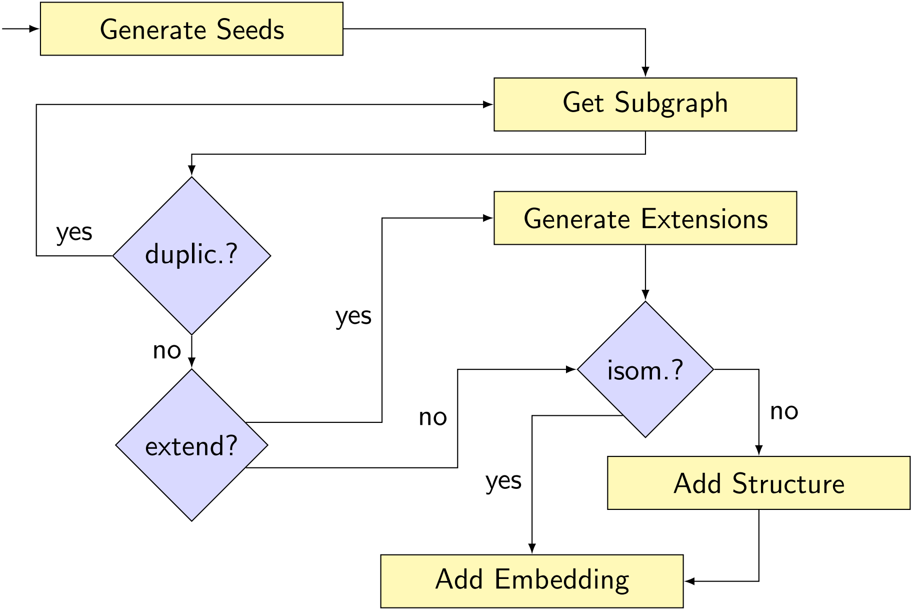

Generate Seeds
Get Subgraph
yes
duplic.?
Generate Extensions
yes
no
extend?

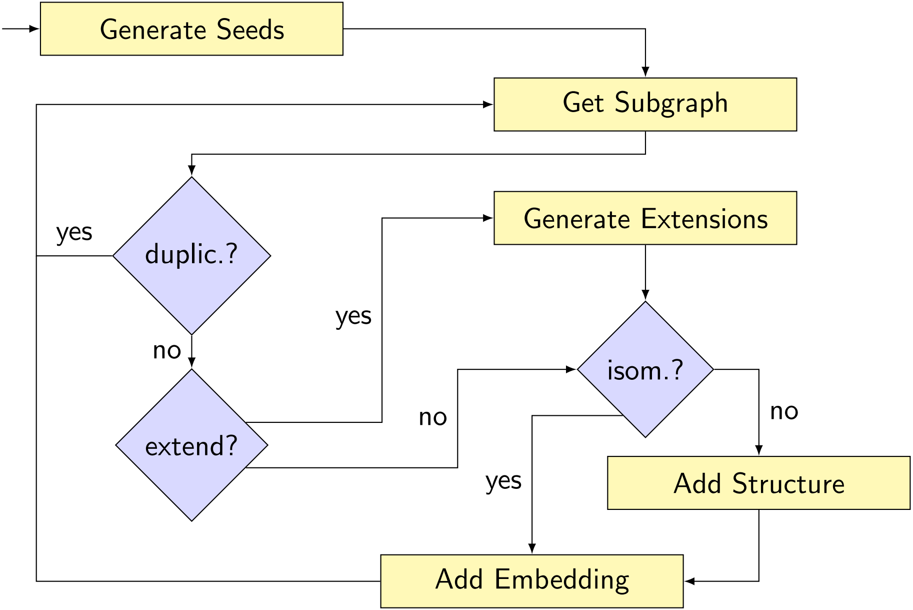

Generate Seeds
Get Subgraph
yes
duplic.?
Generate Extensions
yes
no
isom.?
extend?

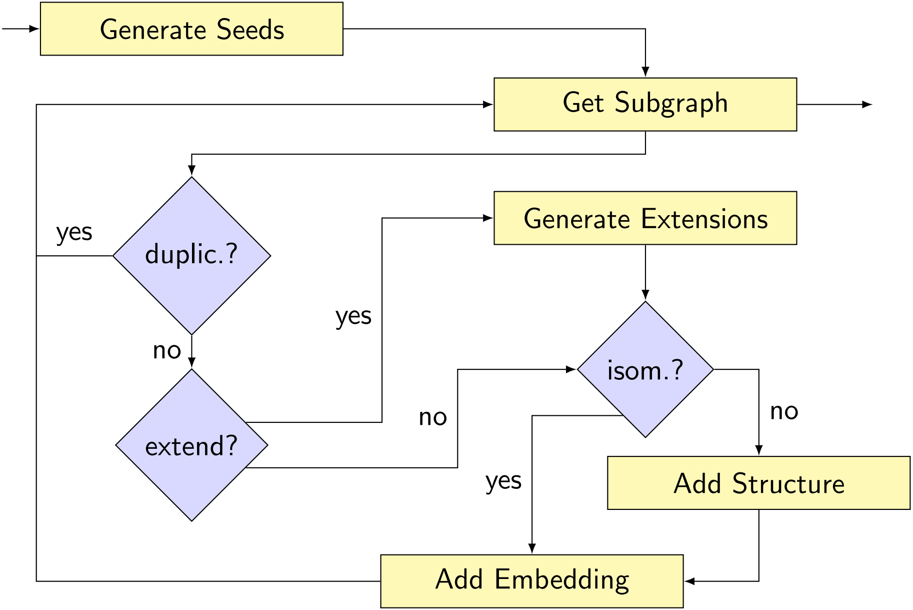

Generate Seeds
Get Subgraph
duplic.?
yes
no
Generate Extensions
extend?
yes
no
isom.?


Generate Seeds
Get Subgraph
yes
duplic.?
no
Generate Extensions
yes
extend?
no
isom.?
no
Add Structure

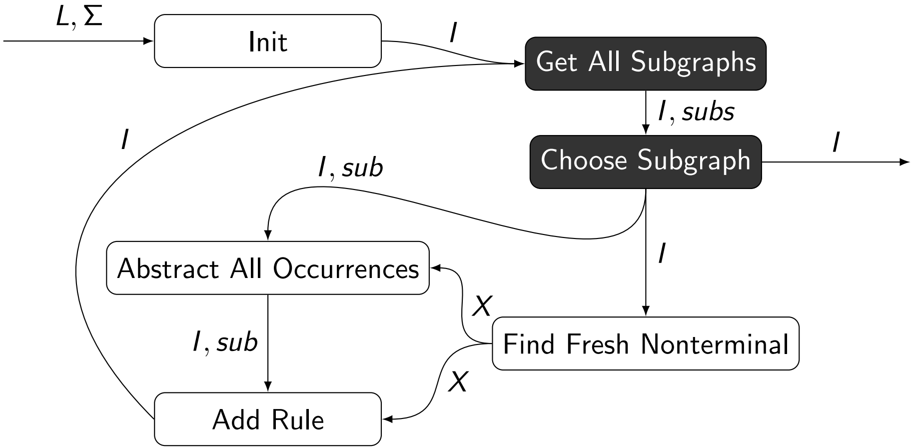

Generate Seeds
Get Subgraph
yes
duplic.?
Generate Extensions
yes
no
isom.?
no
no
extend?
Add Structure
Add Embedding


Generate Seeds
Get Subgraph
yes
duplic.?
no
extend?
yes
Generate Extensions
no
isom.?
yes
no
Add Structure
Add Embedding


Generate Seeds
Get Subgraph
duplic.?
yes
no
Generate Extensions
yes
extend?
no
isom.?
no
yes
Add Structure
Add Embedding

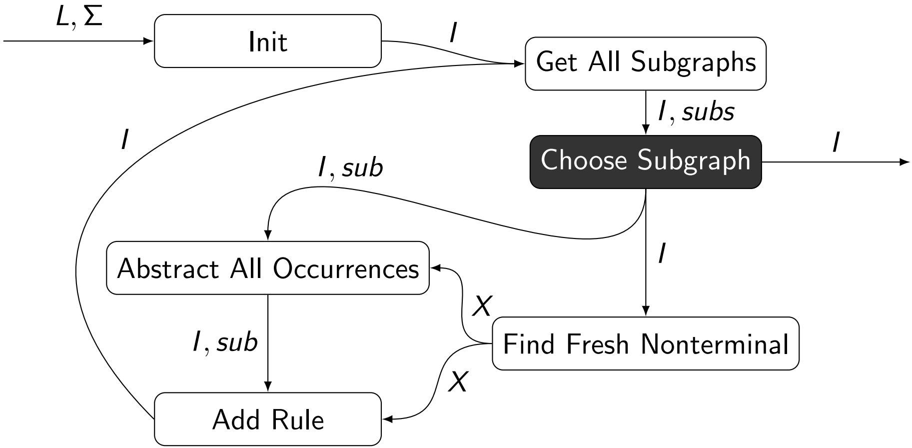

Generate Seeds
Get Subgraph
yes
duplic.?
no
Generate Extensions
yes
extend?
no
isom.?
no
yes
Add Structure
Add Embedding

Overview



L, Σ
Init
I
Get All Subgraphs
$I, subs$
I
Choose Subgraph
I
I, sub
I
Abstract All Occurrences
X
Find Fresh Nonterminal
I, sub
X
Add Rule

Overview



L, Σ
Init
I
Get All Subgraphs
$I, subs$
Choose Subgraph
I
I
I, sub
Abstract All Occurrences
I
X
Find Fresh Nonterminal
I, sub
X
Add Rule

Choosing a Subgraph – Minimum Description Length

Question: Which subgraph to abstract?

Choosing a Subgraph – Minimum Description Length

Question: Which subgraph to abstract?

Answer: Optimization of cost function

Choosing a Subgraph – Minimum Description Length

Question: Which subgraph to abstract?

Answer: Optimization of cost function

Minimum Description Length [Rissanen, 1978]

$$sub = \operatorname{argmin}(cost(X \to H) \qquad\qquad),$$

Choosing a Subgraph – Minimum Description Length

Question: Which subgraph to abstract?

Answer: Optimization of cost function

Minimum Description Length [Rissanen, 1978]

$$sub = \operatorname{argmin}(cost(X \to H) + cost(L')),$$

Choosing a Subgraph – Minimum Description Length

Question: Which subgraph to abstract?

Answer: Optimization of cost function

Minimum Description Length [Rissanen, 1978]

$$sub = \text{argmin}(cost(X \rightarrow H) + cost(L')),$$

where L' is L under the assumption that the rule $X \rightarrow H$ is known.

Description Length

Definition

Cost of a heap configuration G:

$$cost(G) = |V| + |E| + \sum_{e \in E} rk(e)$$

Description Length

Definition

Cost of a heap configuration G:

$$cost(G) = |V| + |E| + \sum_{e \in E} rk(e)$$

Cost of a set of heap configurations L:

$$cost(L) = \sum_{G \in L} cost(G)$$

Description Length

Definition

Cost of a heap configuration G:

$$cost(G) = |V| + |E| + \sum_{e \in E} rk(e)$$

Cost of a set of heap configurations L:

$$cost(L) = \sum_{G \in L} cost(G)$$

Description length of a production rule:

$$cost(X \to H) = 1 + cost(H)$$

Description Length

Definition

Cost of a heap configuration G:

$$cost(G) = |V| + |E| + \sum_{e \in E} rk(e)$$

Cost of a set of heap configurations L:

$$cost(L) = \sum_{G \in L} cost(G)$$

Description length of a production rule:

$$cost(X \to H) = 1 + cost(H)$$

Several definitions possible

Simplification

After a single substitution of H

$$cost(L') = cost(L) - cost(H) + 2 \cdot |ext_H| + 1$$

Simplification

After a single substitution of H

$$cost(L') = cost(L) - cost(H) + 2 \cdot |ext_H| + 1$$

After n substitutions of H

$$cost(L') = cost(L) - n \cdot (cost(H) + 2 \cdot |ext_H| + 1)$$

Simplification

After a single substitution of H

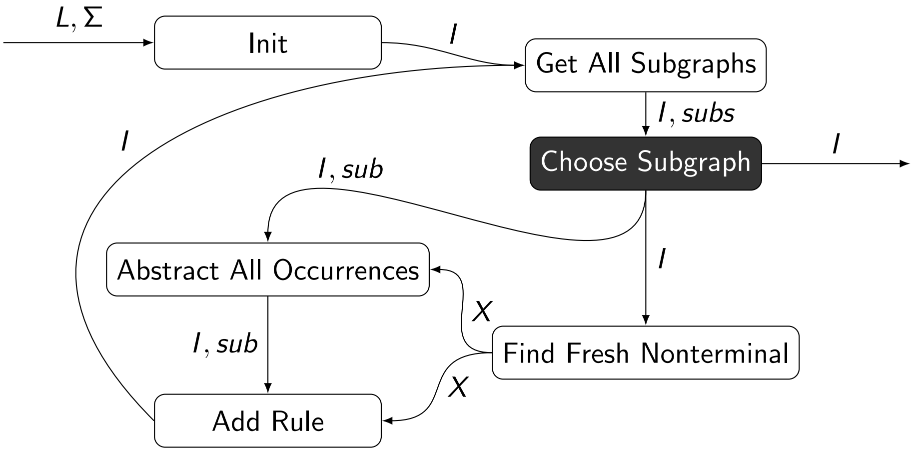
$$cost(L') = cost(L) - cost(H) + 2 \cdot |ext_H| + 1$$

After n substitutions of H

$$cost(L') = cost(L) - n \cdot (cost(H) + 2 \cdot |ext_H| + 1)$$

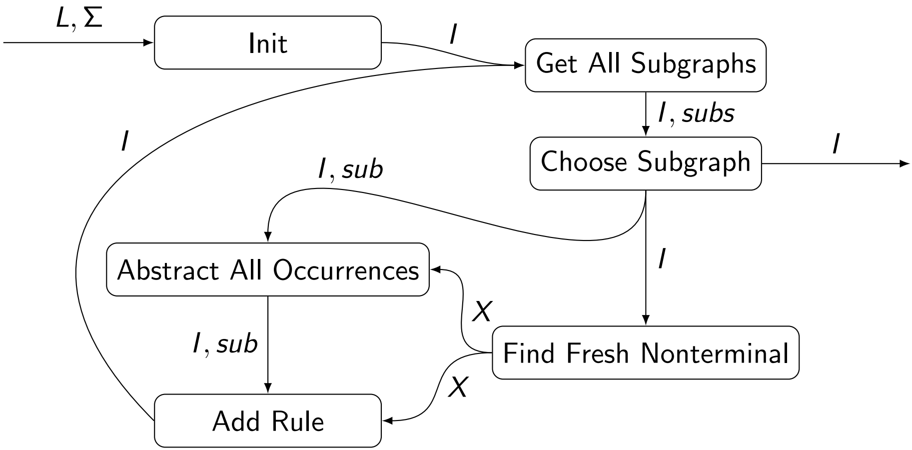
$\Rightarrow$ Computation of cost without actual replacement

Overview

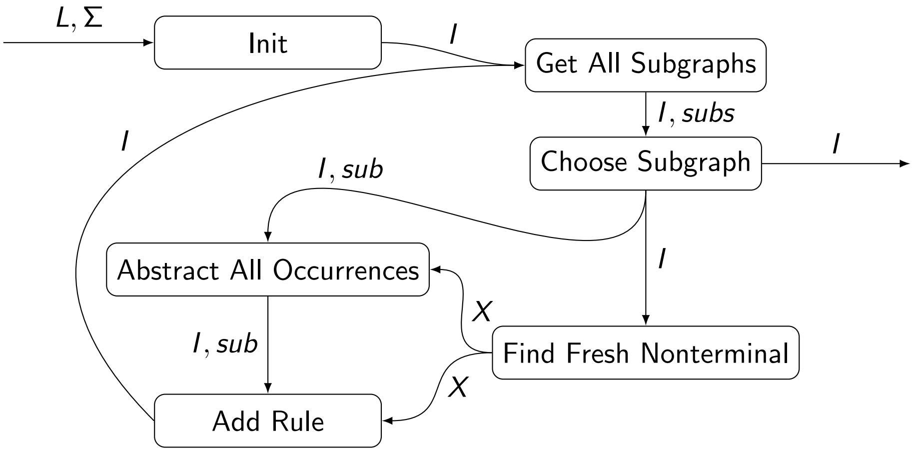


L, Σ
Init
I
Get All Subgraphs
$I, subs$
Choose Subgraph
I
I
I, sub
Abstract All Occurrences
I
X
Find Fresh Nonterminal
I, sub
X
Add Rule

Overview

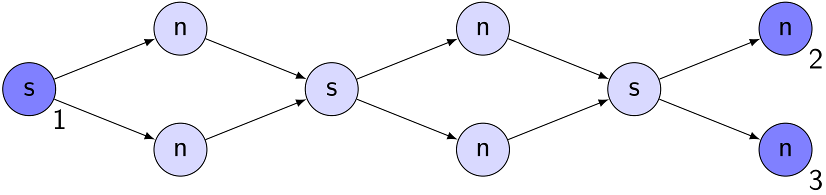

L, Σ
Init
I
Get All Subgraphs
$I, subs$
Choose Subgraph
I
I
I, sub
Abstract All Occurrences
I
X
Find Fresh Nonterminal
I, sub
X
Add Rule

Overview

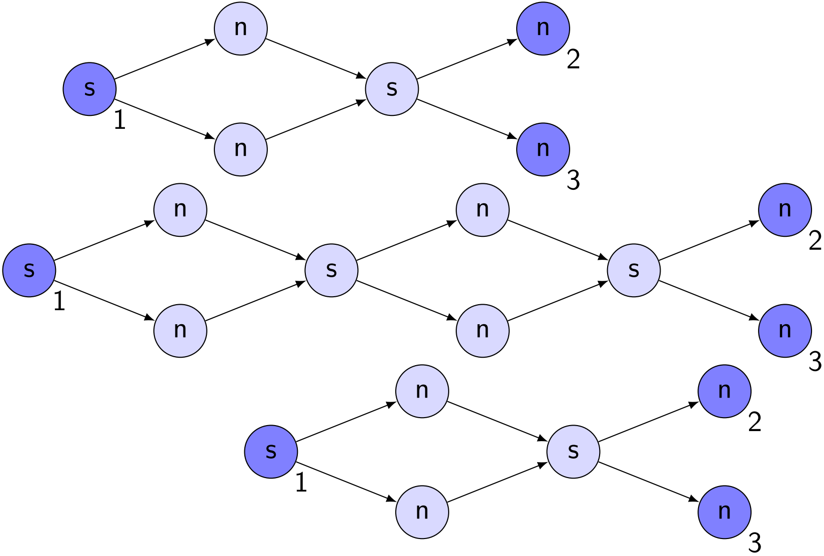


Now: $L(I) \supseteq L$

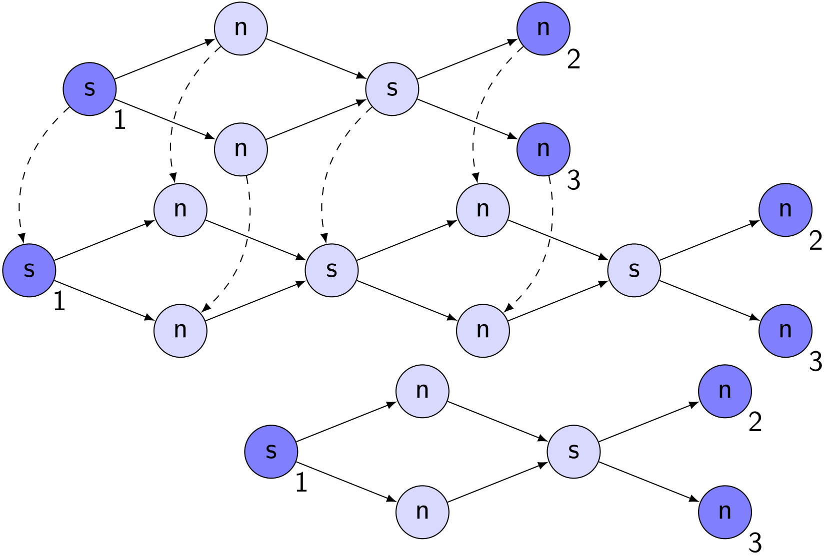
Recursive Structures [Jonyer et al., 2002]



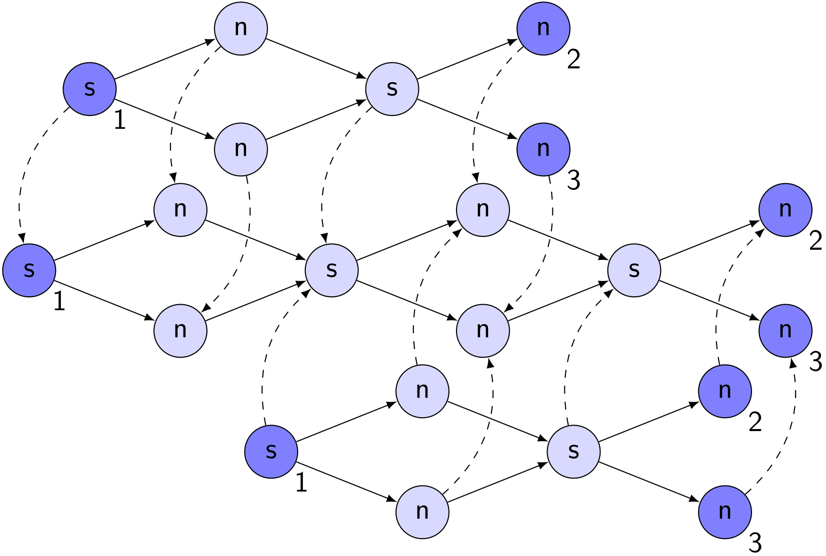
Recursive Structures [Jonyer et al., 2002]

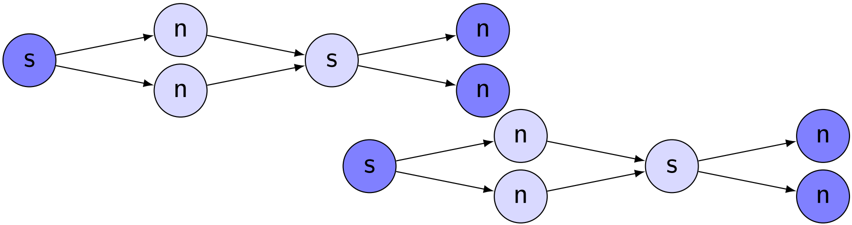


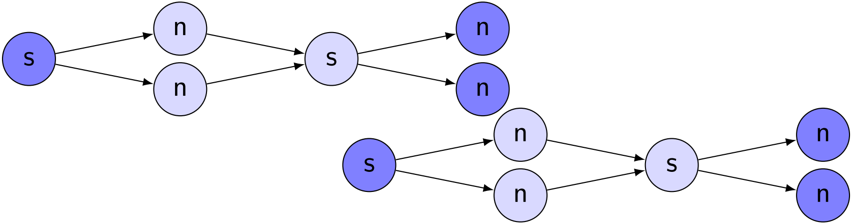
Recursive Structures [Jonyer et al., 2002]



Recursive Structures [Jonyer et al., 2002]

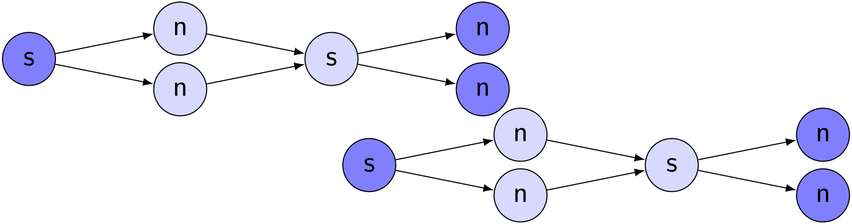



s
n
n
s
n
n
s
n
n
s
n
n



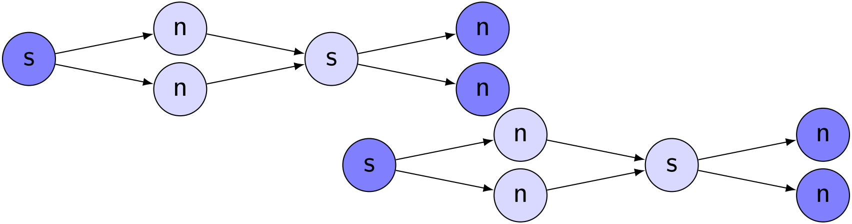
Definition

A structure S is recursive, iff



Definition

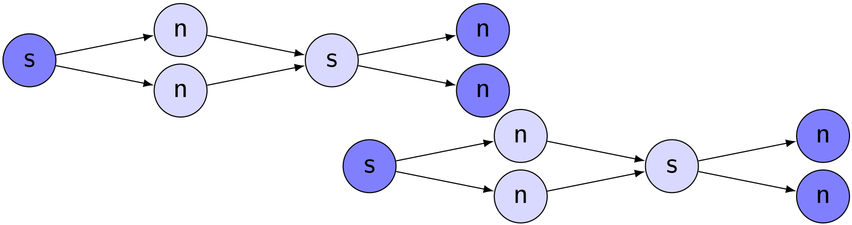
A structure S is recursive, iff there exist two embeddings, such that



Definition

A structure S is recursive, iff there exist two embeddings, such that

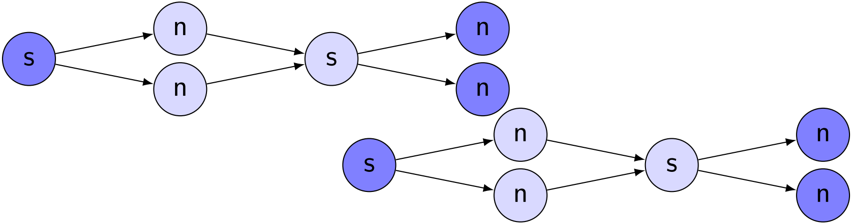
- There are no nonterminal edges attached to the external nodes



Definition

A structure S is recursive, iff there exist two embeddings, such that

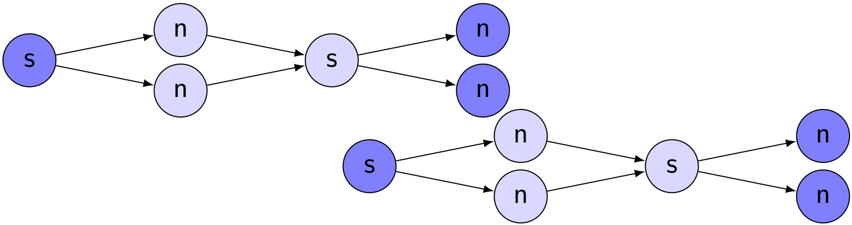
- There are no nonterminal edges attached to the external nodes
- The two embeddings do not share internal nodes



Definition

A structure S is recursive, iff there exist two embeddings, such that

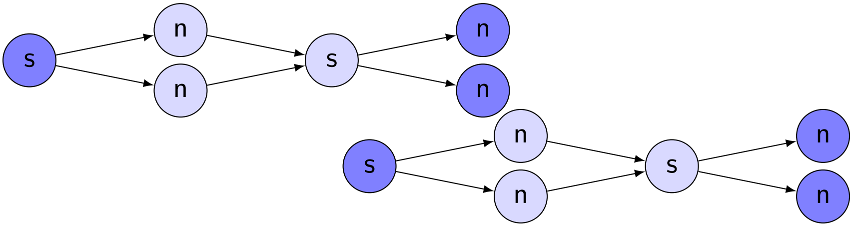
- There are no nonterminal edges attached to the external nodes
- The two embeddings do not share internal nodes
- The external nodes can be partitioned into entry- and exit-nodes



Definition

A structure S is recursive, iff there exist two embeddings, such that

- There are no nonterminal edges attached to the external nodes
- The two embeddings do not share internal nodes
- The external nodes can be partitioned into entry- and exit-nodes
- There are no incoming pointers to the entry-nodes
- There are no outgoing pointers from the exit-nodes



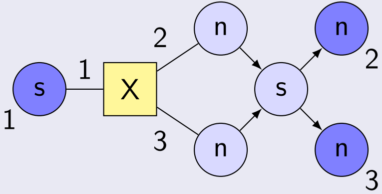
Definition

A structure S is recursive, iff there exist two embeddings, such that

- There are no nonterminal edges attached to the external nodes
- The two embeddings do not share internal nodes
- The external nodes can be partitioned into entry- and exit-nodes
- There are no incoming pointers to the entry-nodes
- There are no outgoing pointers from the exit-nodes
- The exit-nodes of one embedding can be reached from the entry-nodes of the other one

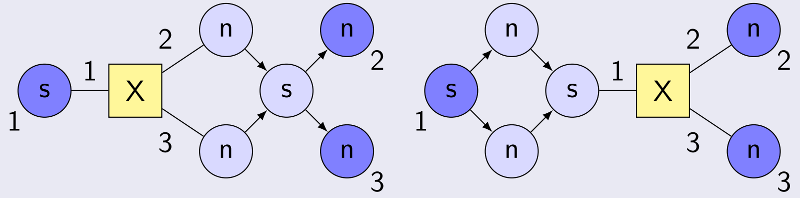
Recursive Rules added to the Grammar

Concatenation Rules



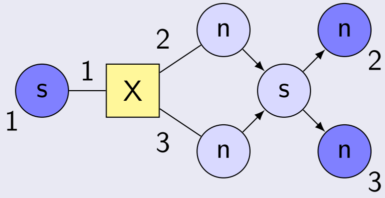
Recursive Rules added to the Grammar

Concatenation Rules

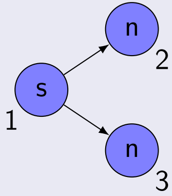


Recursive Rules added to the Grammar

Concatenation Rules

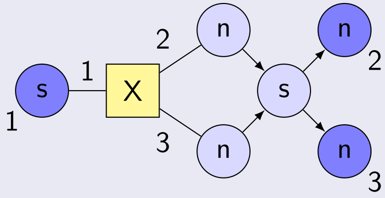


Finalization Rule

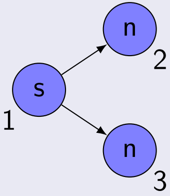


Recursive Rules added to the Grammar

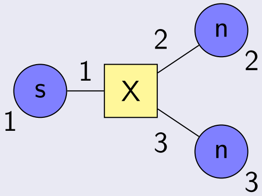
Concatenation Rules



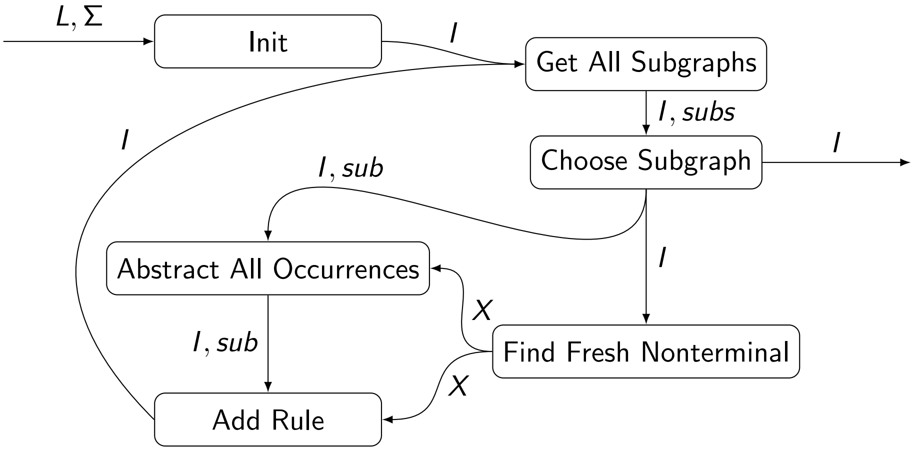
Finalization Rule



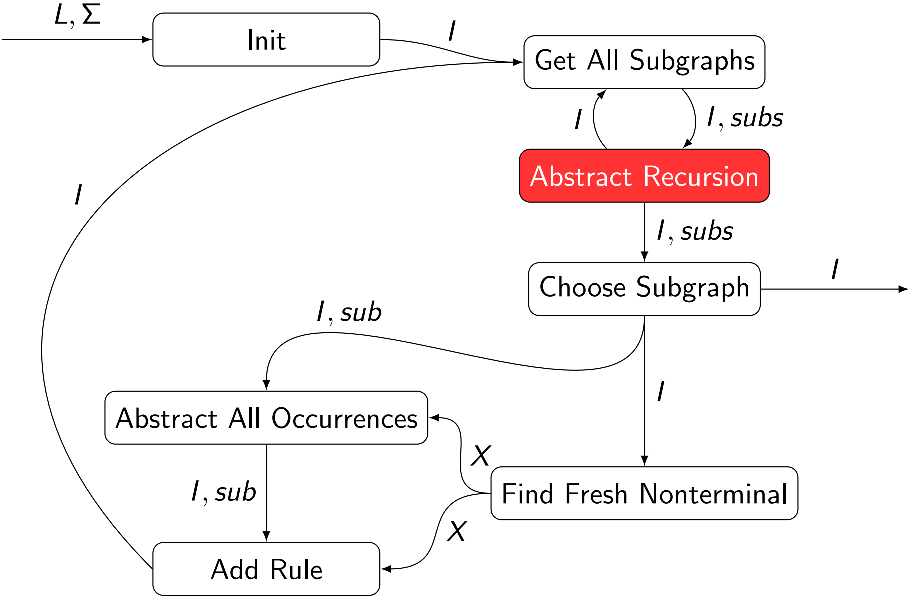
Abstraction in Graph



Complete Algorithm



Complete Algorithm



Singly Linked List

Input: Singly linked lists with 25 to 200 nodes

Singly Linked List

Input: Singly linked lists with 25 to 200 nodes

Nodes	Subgraphs [ms]	Complete [ms]
25	90	102
50	285	305
75	437	500
100	642	682
125	1 001	1 040
150	1 455	1 526
175	1 884	2 000
200	2 895	3 028

Singly Linked Circular List

Input: Singly linked circular lists with 25 to 200 nodes

Singly Linked Circular List

Input: Singly linked circular lists with 25 to 200 nodes

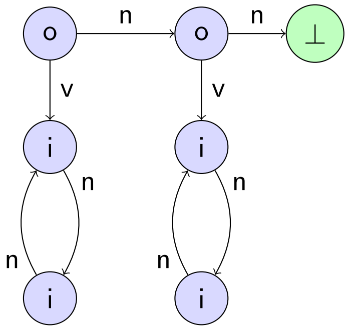
Nodes	Subgraphs [ms]	Complete [ms]
25	75	90
50	261	281
75	451	464
100	680	658
125	979	1 032
150	1 465	1 511
175	1 889	1 933
200	2 805	2 995

Singly Linked Nested List

Input: Singly linked nested lists

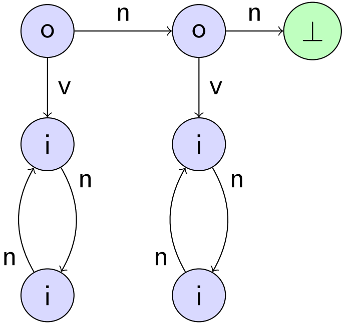
Singly Linked Nested List

Input: Singly linked nested lists



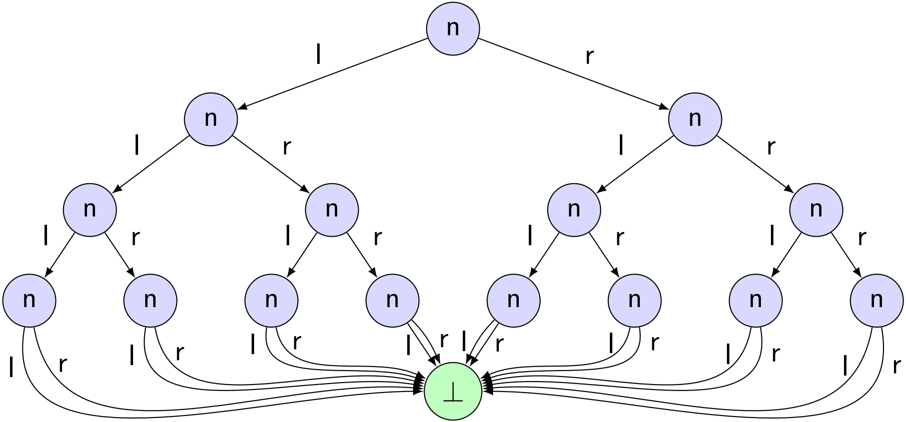
Singly Linked Nested List

Input: Singly linked nested lists



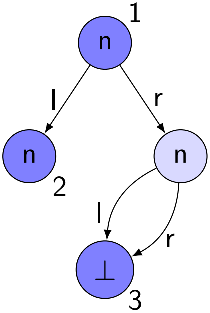
Outer	Inner	Inner [ms]	Outer [ms]
2	2	9	8
2	4	19	5
4	2	31	16
4	4	394	11
4	5	1 280	20
5	2	96	23
5	4	2 701	22
5	5	51 608	23

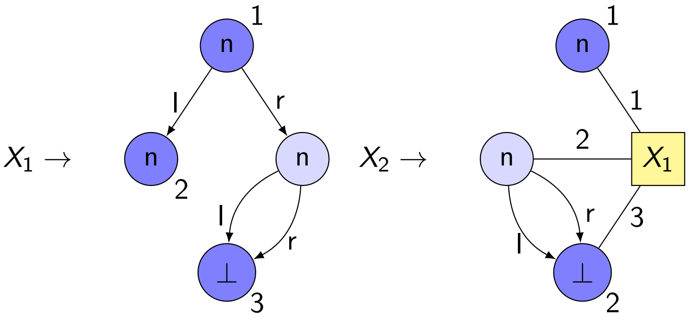
Binary Tree



Binary Tree – Rules

$X_1 \rightarrow$




$X_1 \rightarrow$
1
n
l
r
n
2
n
l
r
⊥
3
$X_2 \rightarrow$
1
n
1
2
n
X_1
r
3
l
⊥
2

Summary

- Definition: Heap Configurations and Heap Abstraction Grammars

Summary

- Definition: Heap Configurations and Heap Abstraction Grammars
- Problem: Generate grammar from set of Heap Configurations

Summary

- Definition: Heap Configurations and Heap Abstraction Grammars
- Problem: Generate grammar from set of Heap Configurations
- Solution: Abstract subgraphs step by step
  1. Find recursive subgraphs
  2. Abstract subgraph with greatest gain in description length

Summary

- Definition: Heap Configurations and Heap Abstraction Grammars
- Problem: Generate grammar from set of Heap Configurations
- Solution: Abstract subgraphs step by step
    1. Find recursive subgraphs
    2. Abstract subgraph with greatest gain in description length
- Backed by experimental results

Further Research

- Subgraph enumeration avoids certain graphs

Further Research

- Subgraph enumeration avoids certain graphs
  - Efficiently enumerate subgraphs with unconnected internal nodes

Further Research

- Subgraph enumeration avoids certain graphs
  - Efficiently enumerate subgraphs with unconnected internal nodes
- Recursion only works for simply concatenations

Further Research

- Subgraph enumeration avoids certain graphs
    - Efficiently enumerate subgraphs with unconnected internal nodes
- Recursion only works for simply concatenations
    - Allow multiple sets of entry- and exit-points

Thank you for your attention

Heinen, J., Jansen, C., Katoen, J.-P., and Noll, T. (tbp).
Juggrnaut: Using Graph Grammars for Abstracting Unbounded Heap Structures.

Heinen, J., Noll, T., and Rieger, S. (2009).
Juggrnaut: Graph Grammar Abstraction for Unbounded Heap Structures.
In Johnsen, E. B. and Stolz, V., editors, *Harnessing Theories for Tool Support in Software, Preliminary Proceedings*, pages 53–67. United Nations University - International Institute for Software Technology.

Jansen, C. (2010).
Konstruktion und Inferenz von Hypergraphabstraktionsgrammatiken.
Diplomarbeit, RWTH Aachen University.

Jonyer, I., Holder, L. B., and Cook, D. J. (2002).
Concept Formation Using Graph Grammars.
In *Proceedings of the KDD Workshop on Multi-Relational Data Mining*.

Rissanen, J. (1978).
Modeling by shortest data description.
Automatica, 14(5):465–471.