Factorization of Synchronous Context-Free Grammars in Linear Time

Hao Zhang and Daniel Gildea

Computer Science Department

University of Rochester

Outline

- Introduction to Synchronous Context Free Grammar (SCFG)
- Permutation trees
- A shift-reduce framework
- A linear time shift-reduce algorithm
- Experiments on alignment analysis

Notions for SCFG

- A generic n-ary SCFG rule is written as

$$X \longrightarrow X_1^{(1)} ... X_n^{(n)}, \ X_{\pi(1)}^{(\pi(1))} ... X_{\pi(n)}^{(\pi(n))}$$

where each X_i is a variable which can take the value of any nonterminal in the grammar.

- For example, the 3-ary rule $S \longrightarrow \left[\begin{array}{c|c|c} & & A \\ \hline A & & \\ \hline & B & \end{array}\right]$ can be written as

$$S \longrightarrow A^{(1)} B^{(2)} A^{(3)}, \ B^{(2)} A^{(1)} A^{(3)}$$

where $\pi = (2, 1, 3)$.

Factorization to Reduce SCFG Parsing Complexity

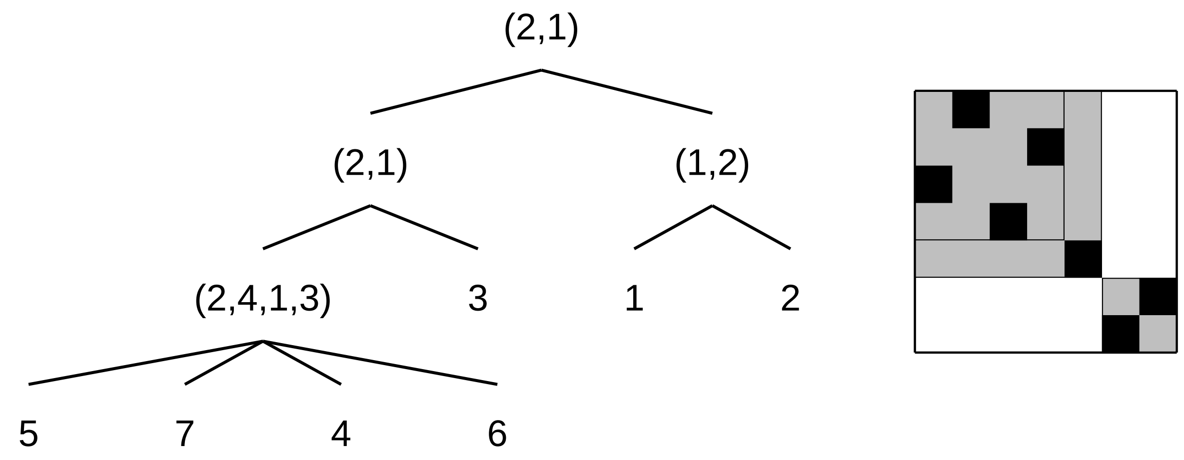
It is possible to recursively decompose SCFG rules. For example,

$$[\, X \longrightarrow A^{(1)}B^{(2)}C^{(3)}D^{(4)}E^{(5)}F^{(6)}G^{(7)},$$

$$X \longrightarrow E^{(5)}G^{(7)}D^{(4)}F^{(6)}C^{(3)}A^{(1)}B^{(2)}\,]$$

is decomposable by analyzing the structure of

$\pi = (5, 7, 4, 6, 3, 1, 2)$:



Factorization to Reduce SCFG Parsing Complexity

$$[\, X \longrightarrow X_1^{(1)} X_2^{(2)}, \;\; X \longrightarrow X_2^{(2)} X_1^{(1)} \,]$$

$$[\, X_1 \longrightarrow A^{(1)} B^{(2)}, \;\; X_1 \longrightarrow A^{(1)} B^{(2)} \,]$$

$$[\, X_2 \longrightarrow C^{(1)} X_3^{(2)}, \;\; X_2 \longrightarrow X_3^{(2)} C^{(1)} \,]$$

$$[\, X_3 \longrightarrow D^{(1)} E^{(2)} F^{(3)} G^{(4)},$$

$$X_3 \longrightarrow E^{(2)} G^{(4)} D^{(1)} F^{(3)} \,]$$

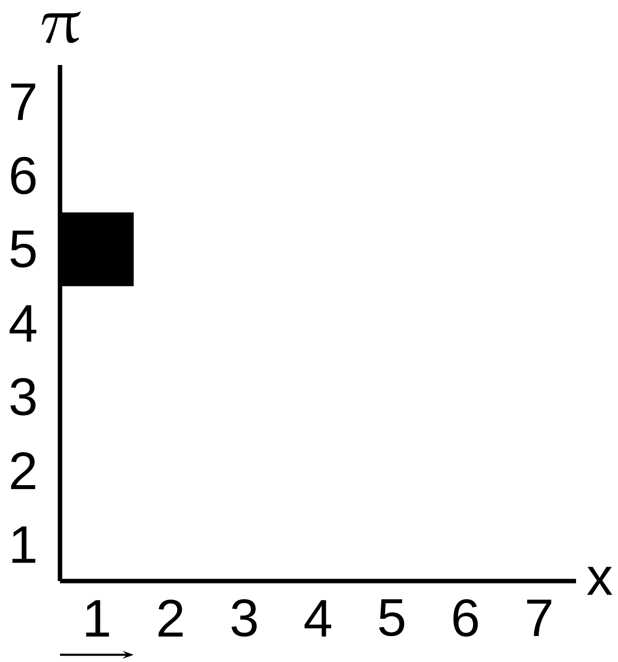
Notions for Parsing Permutations

- **permuted sequence**: such as (5, 7, 4, 6, 3, 1, 2), (5, 7, 4, 6), and (1, 2). If a permuted sequence has been found, we can reduce it to a subsequence (block) of [min...max], such as [4...7] and [1...2]. A block serves as a *pebble* in latter reductions.
- **permutation tree**: a hierarchy of permuted sequences.
- k**-arizer**: parse a permutation into a permutation tree with the maximal fanout of any node as k.

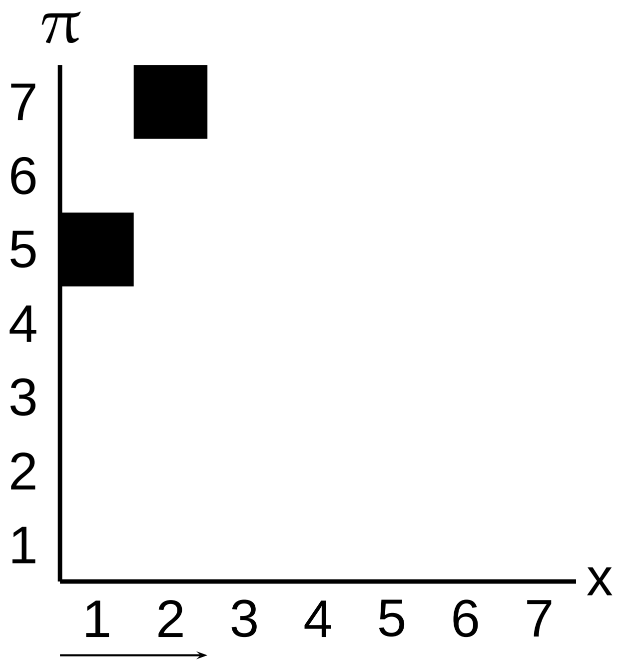
Shift-Reduce k-arizer

1. Shift the next number in the input permutation onto the stack.
2. Repeatedly try the 2-ary, 3-ary, ..., k-ary permutations to reduce the subsequences on the top of the stack to one long subsequence.
3. If there are remaining numbers in the input permutation, go to 1.

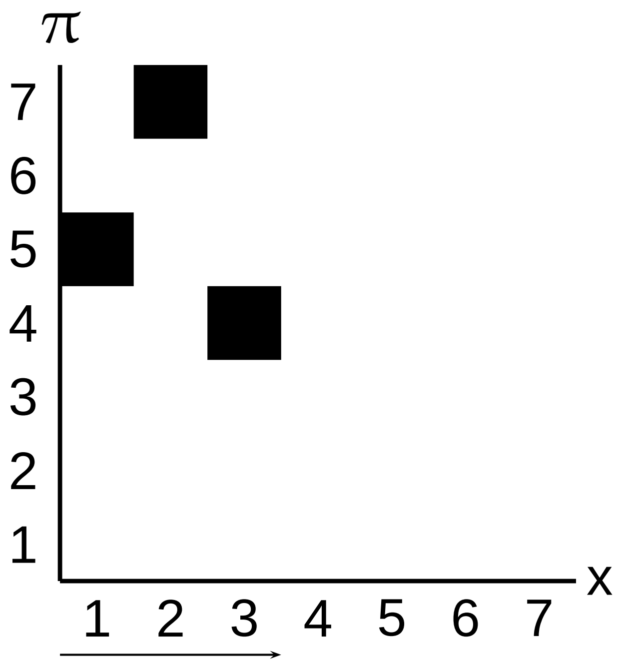
Example Execution Trace



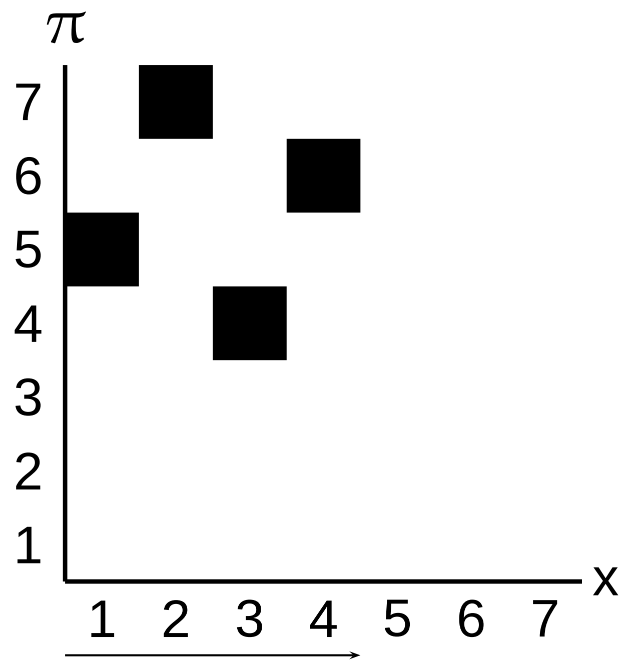
Example Execution Trace



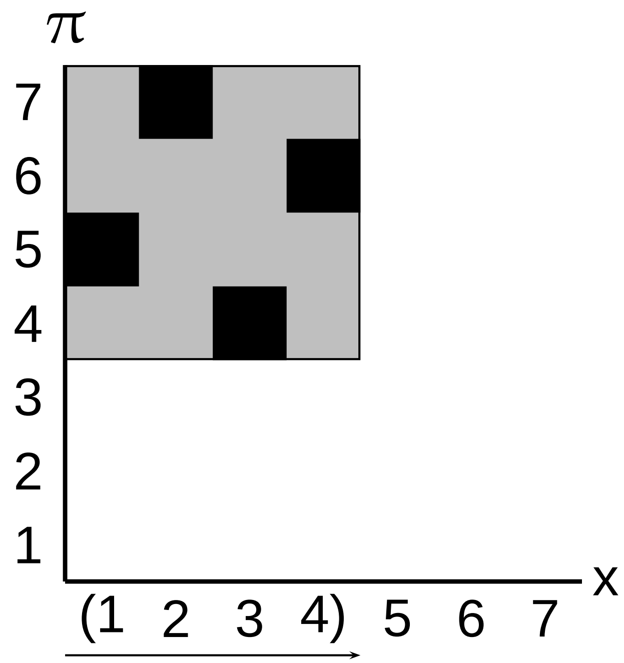
Example Execution Trace



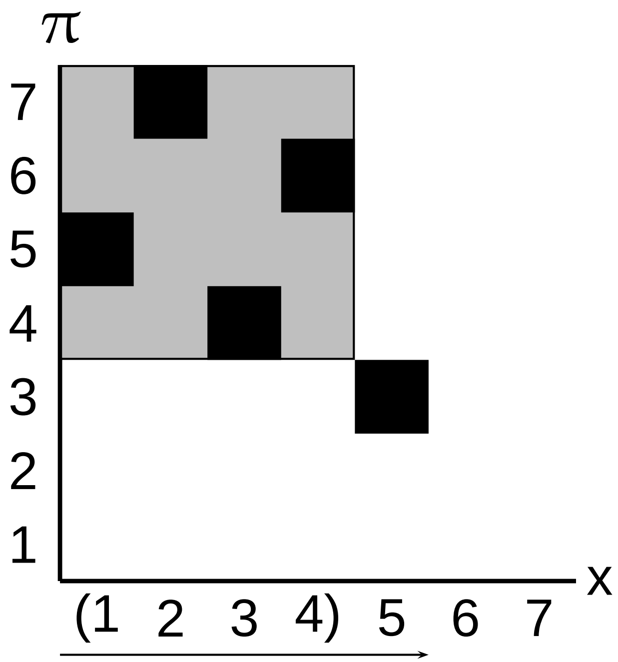
Example Execution Trace



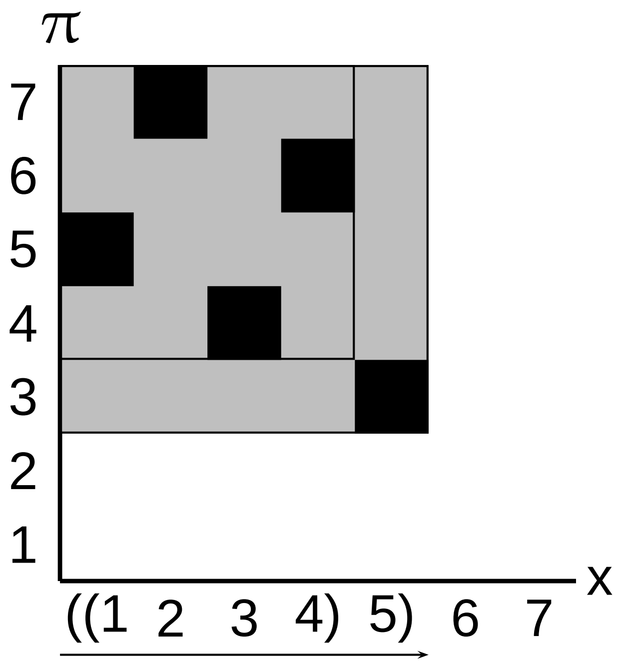
Example Execution Trace



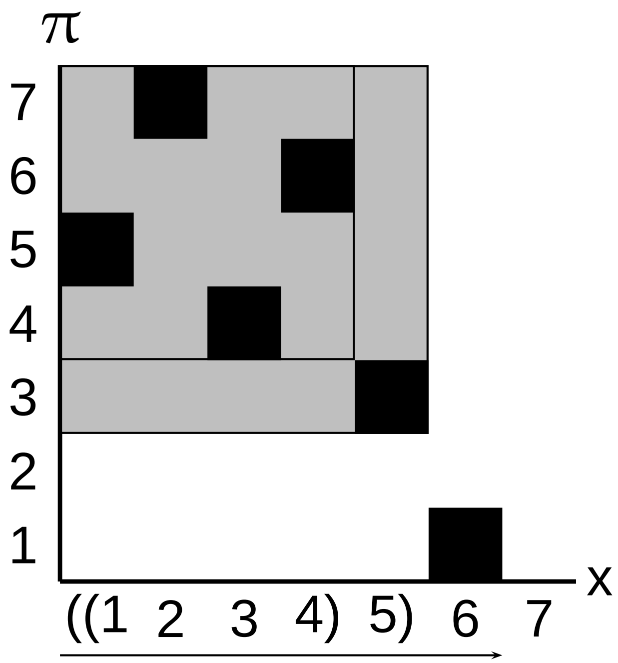
Example Execution Trace



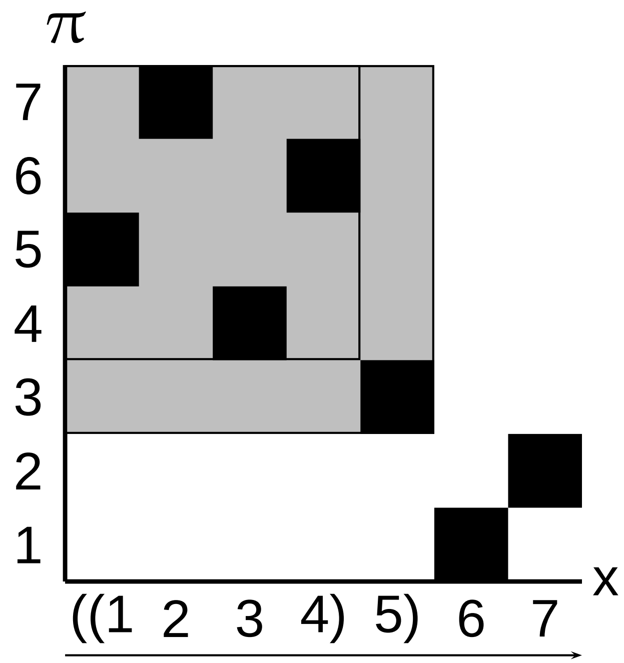
Example Execution Trace



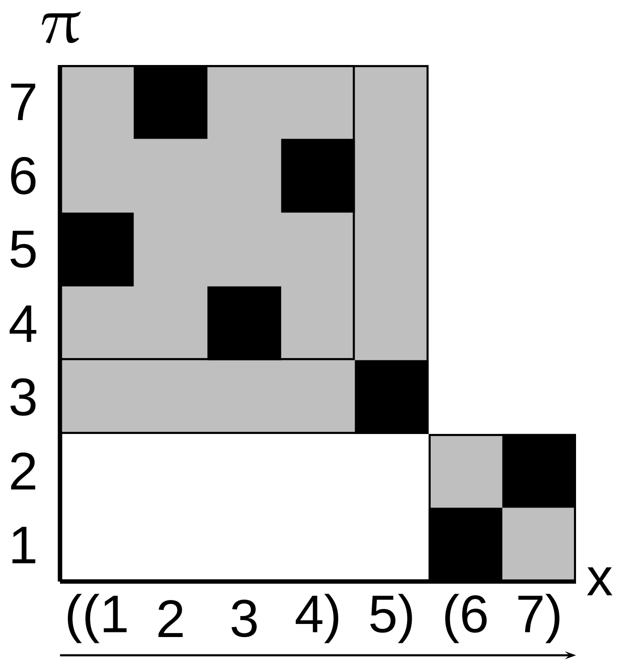
Example Execution Trace



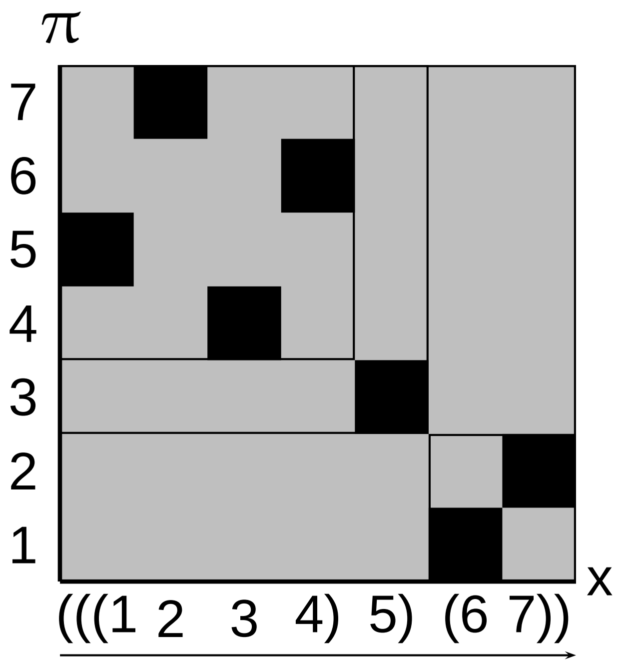
Example Execution Trace



Example Execution Trace



Example Execution Trace



Mathematical Formulation of the Problem

We define a function whose value indicates the reducibility of each pair of positions (x, y) $(1 \leq x \leq y \leq n)$:

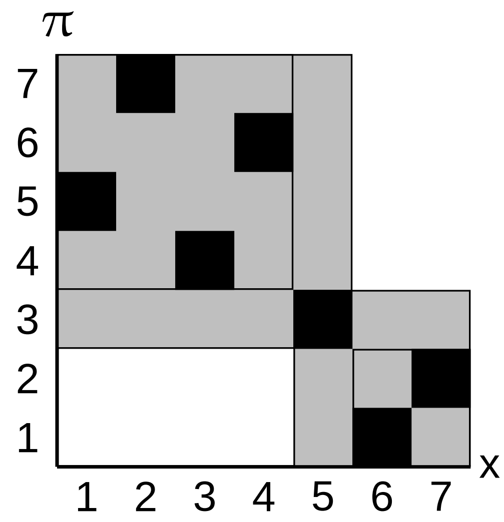
$$f(x, y) = u(x, y) - l(x, y) - (y - x)$$

where

$$l(x, y) = \min_{i \in [x,y]} \pi(i)$$
$$u(x, y) = \max_{i \in [x,y]} \pi(i)$$

l records the minimum of the numbers that are permuted to from the positions in the region $[x, y]$. u records the maximum.

The All Common Interval Problem

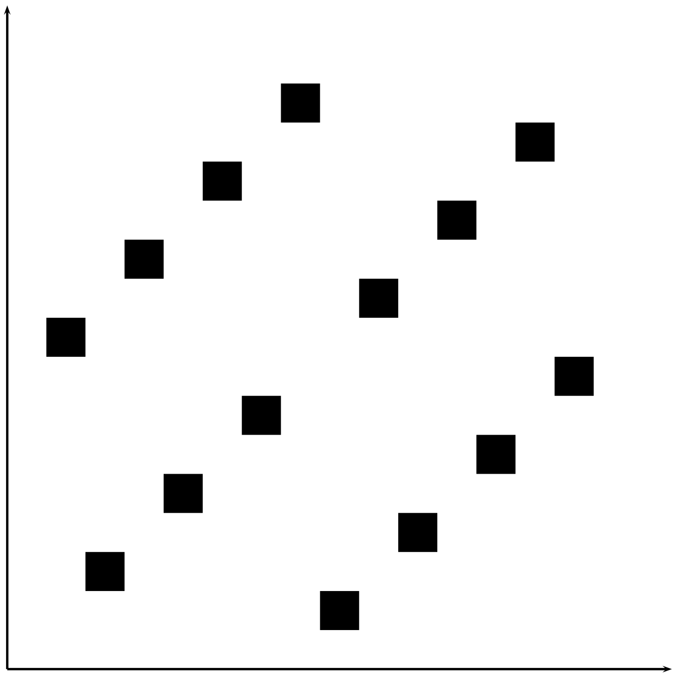


- Uno and Yagiura (2000) devised a sweeping-scan algorithm of $O(n + K)$, where K is the number of common intervals.
- Output possibly contains overlapping blocks. ($K = O(n^2)$)
- We modify the algorithm into a shift-reduce algorithm that outputs recursive common intervals.

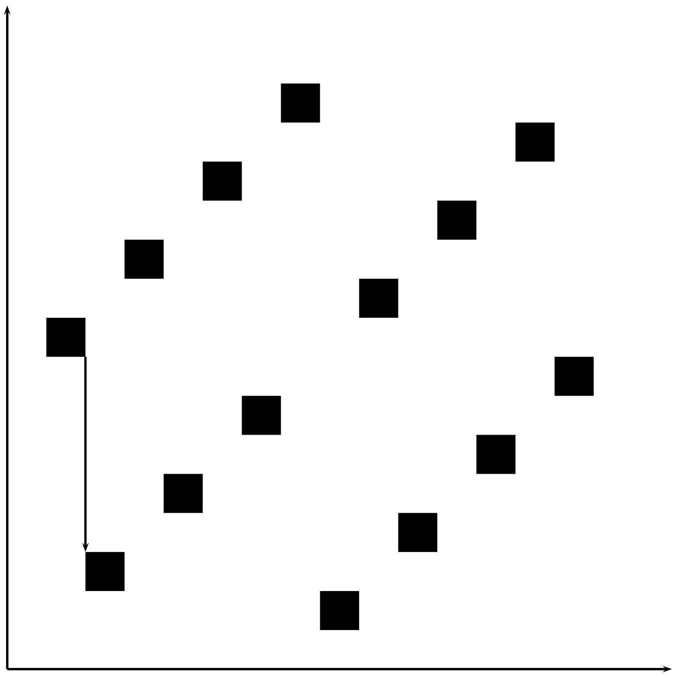
A Linear Time Factorization Algorithm

- examines $f(x, y) = 0$ on (x, y) pairs, left-to-right on y in the outer loop, and right-to-left on x in the inner loop.
- *eliminates "bad" candidate* x*'s at the first time seen.*
- O(n) reducible spans.
- O(n) operations.

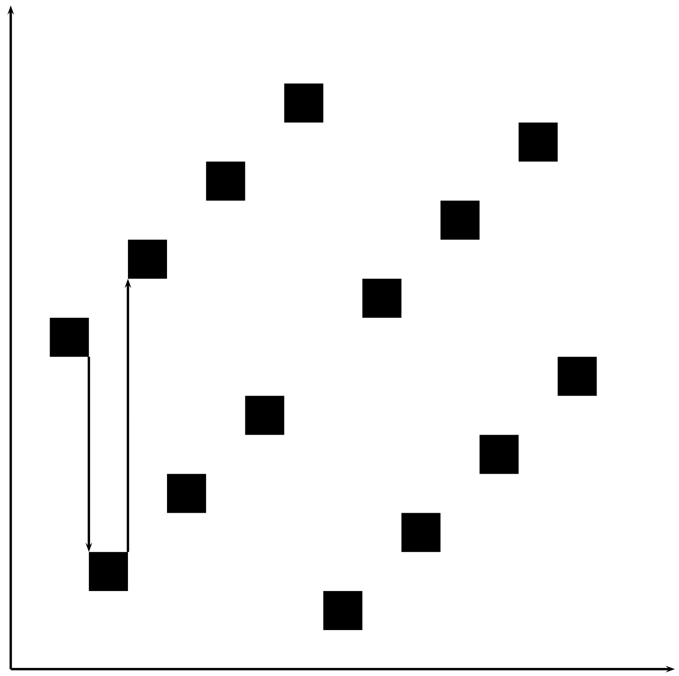
Candidate Elimination Elaborated



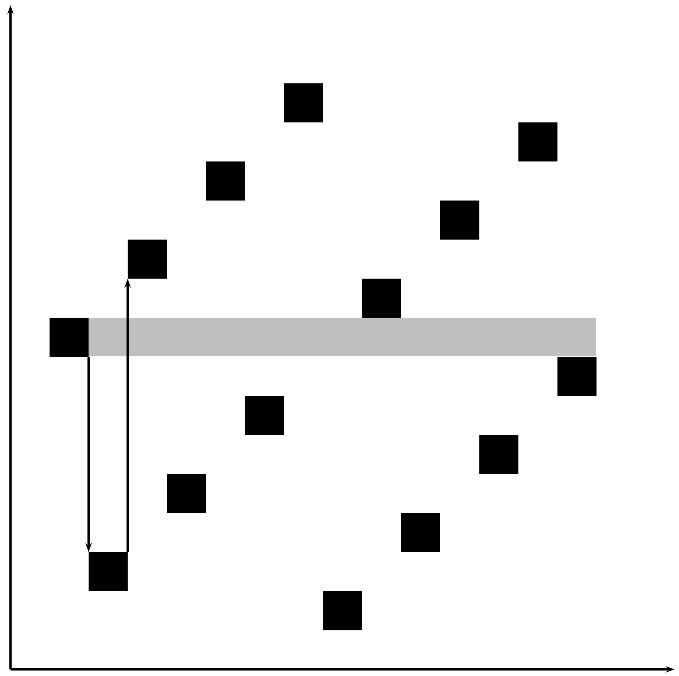
Candidate Elimination Elaborated



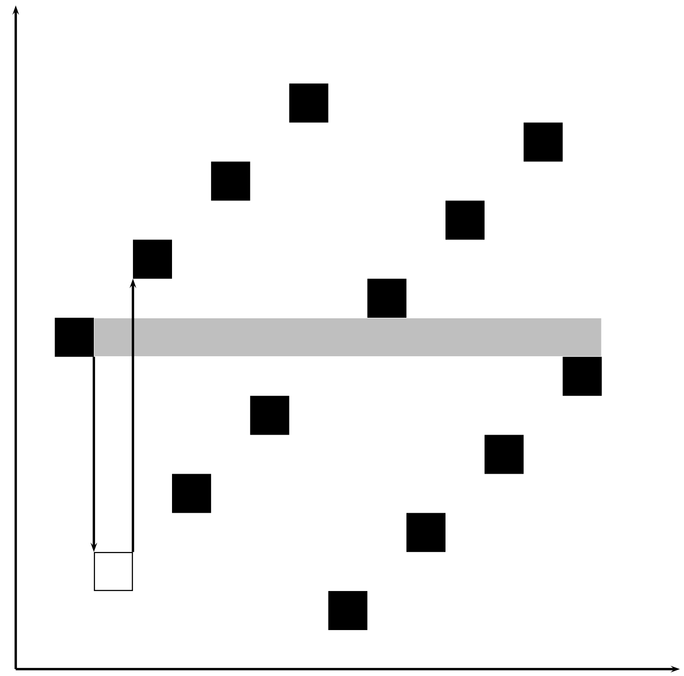
Candidate Elimination Elaborated



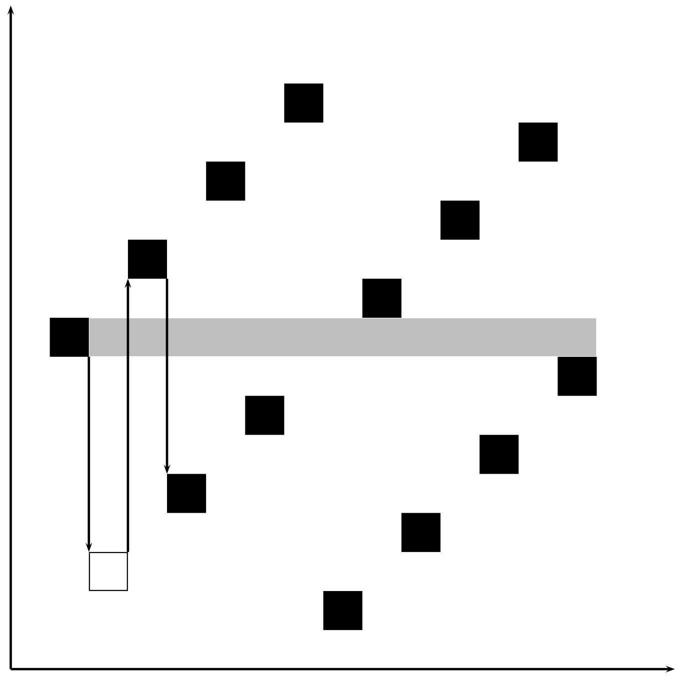
Candidate Elimination Elaborated



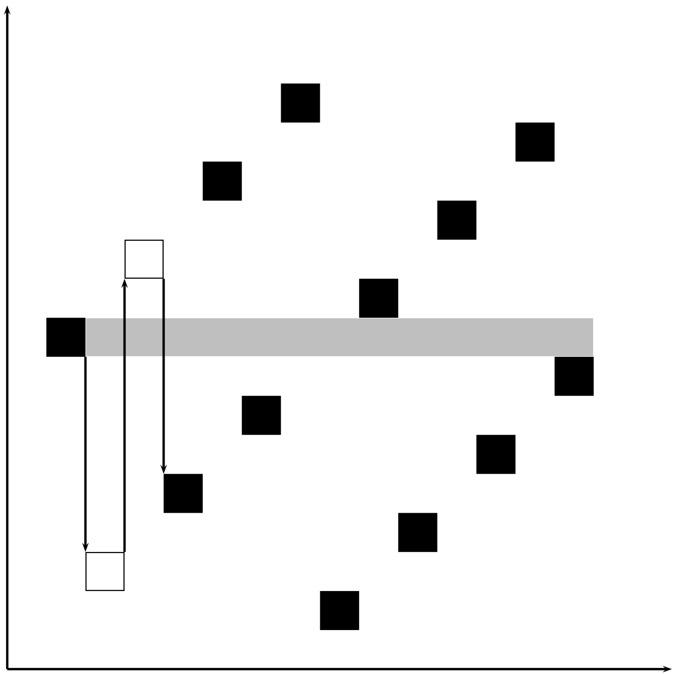
Candidate Elimination Elaborated



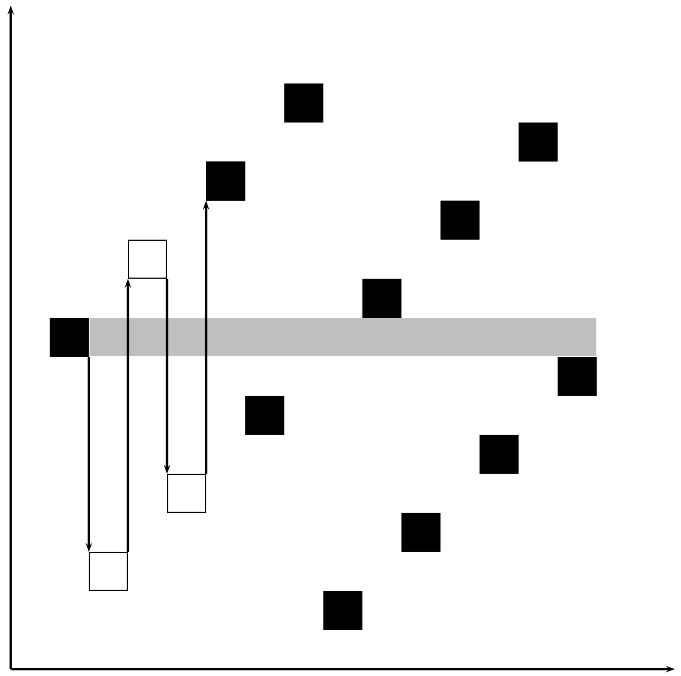
Candidate Elimination Elaborated



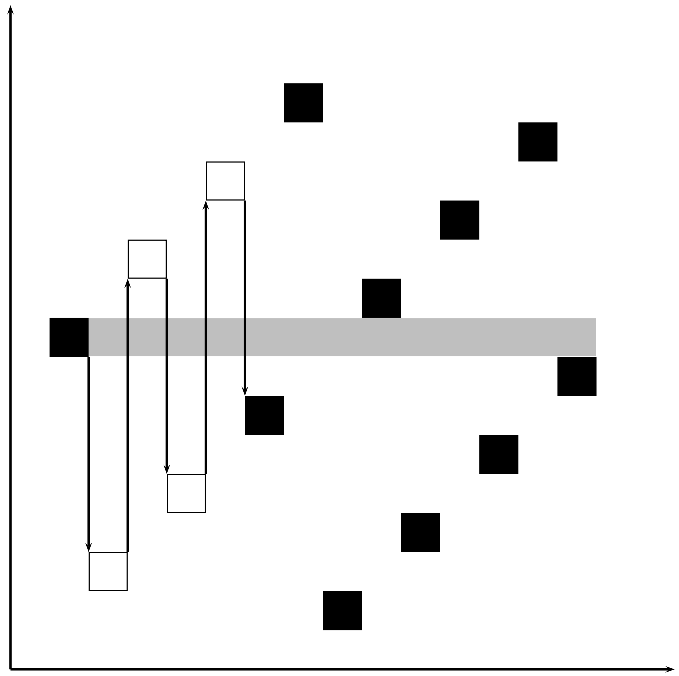
Candidate Elimination Elaborated



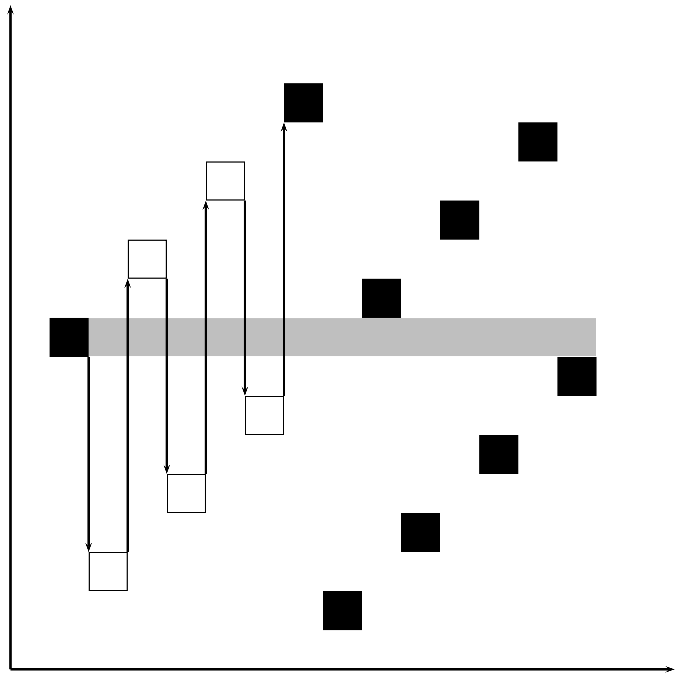
Candidate Elimination Elaborated



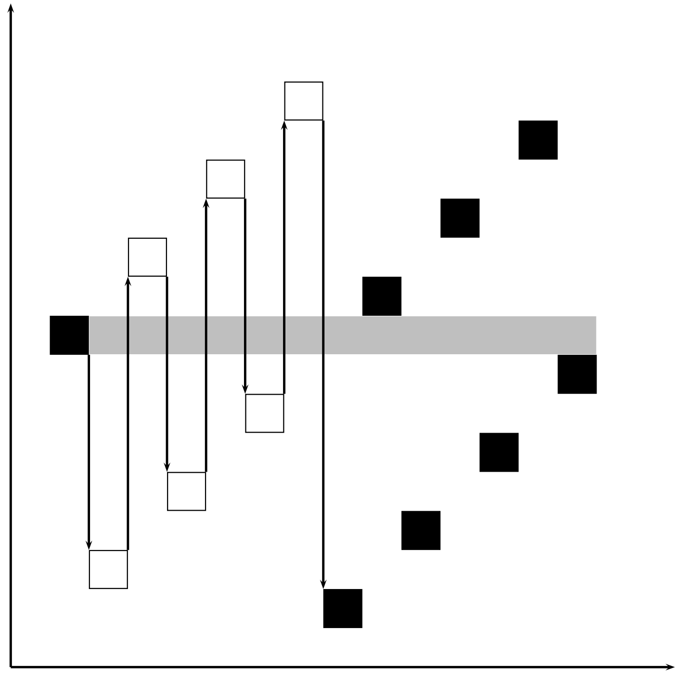
Candidate Elimination Elaborated



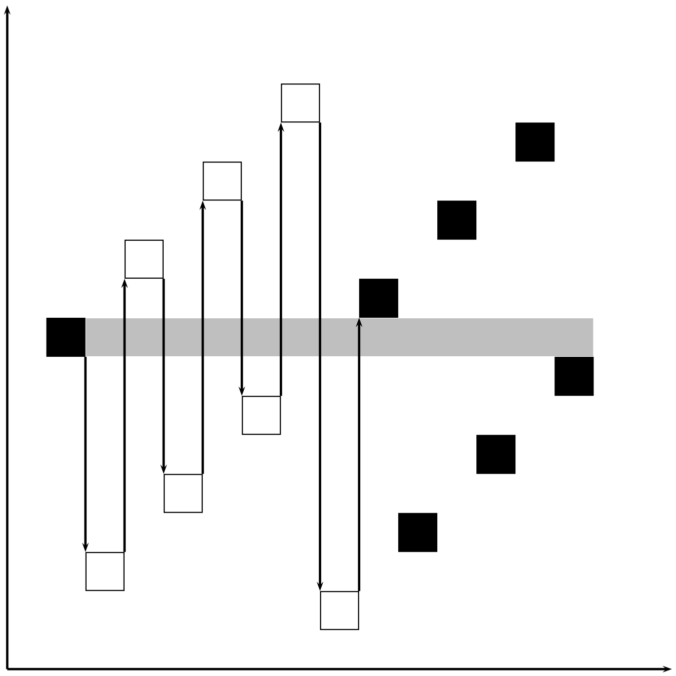
Candidate Elimination Elaborated



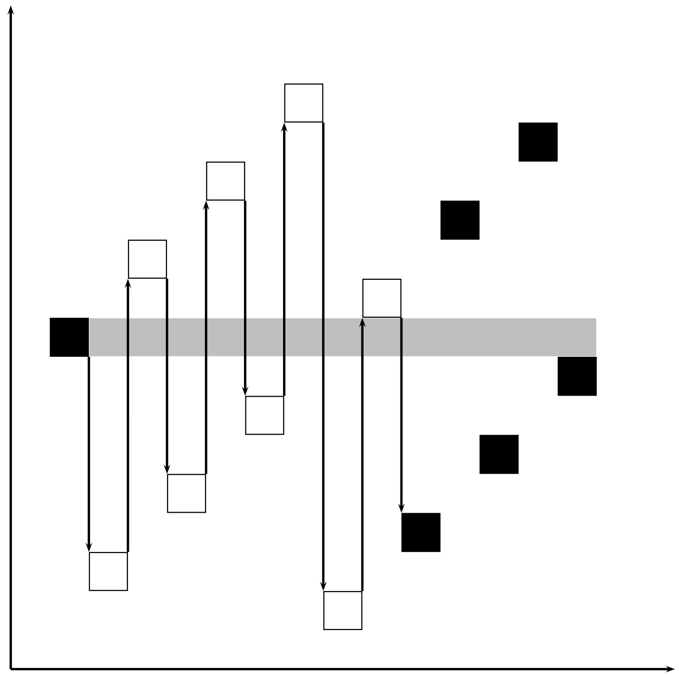
Candidate Elimination Elaborated



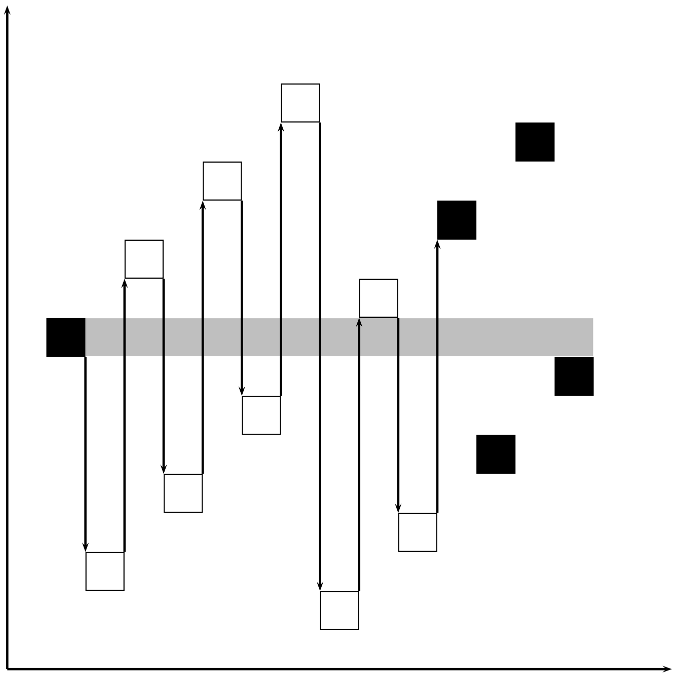
Candidate Elimination Elaborated



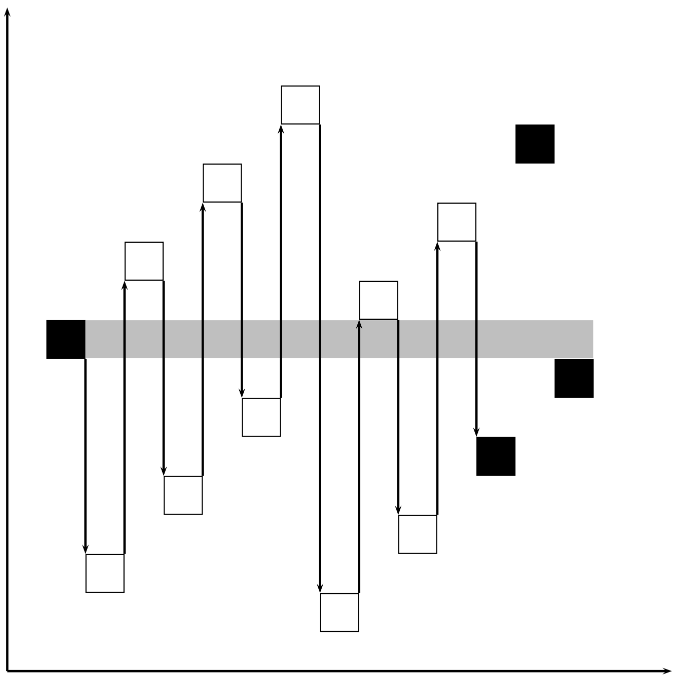
Candidate Elimination Elaborated



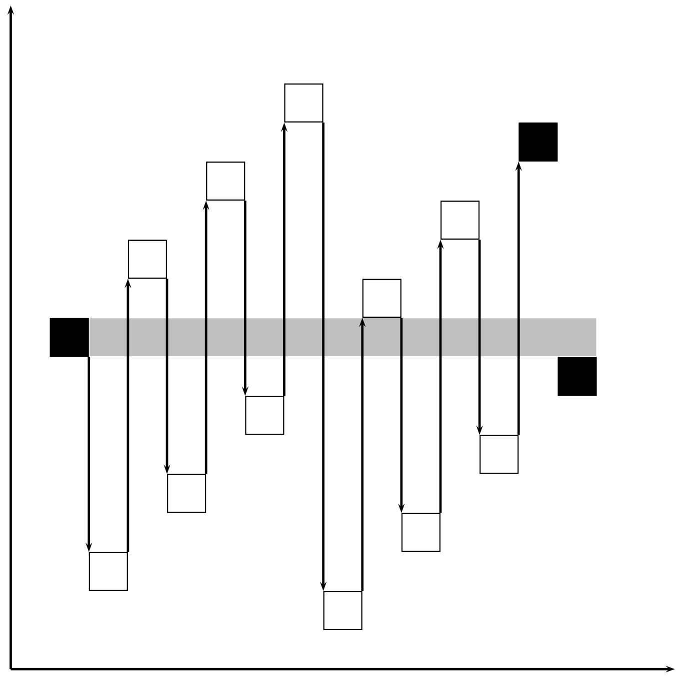
Candidate Elimination Elaborated



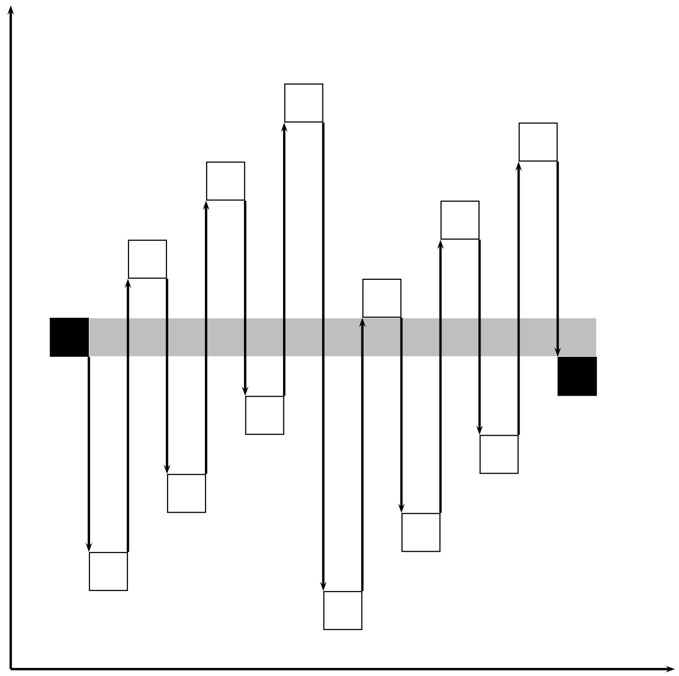
Candidate Elimination Elaborated



Candidate Elimination Elaborated

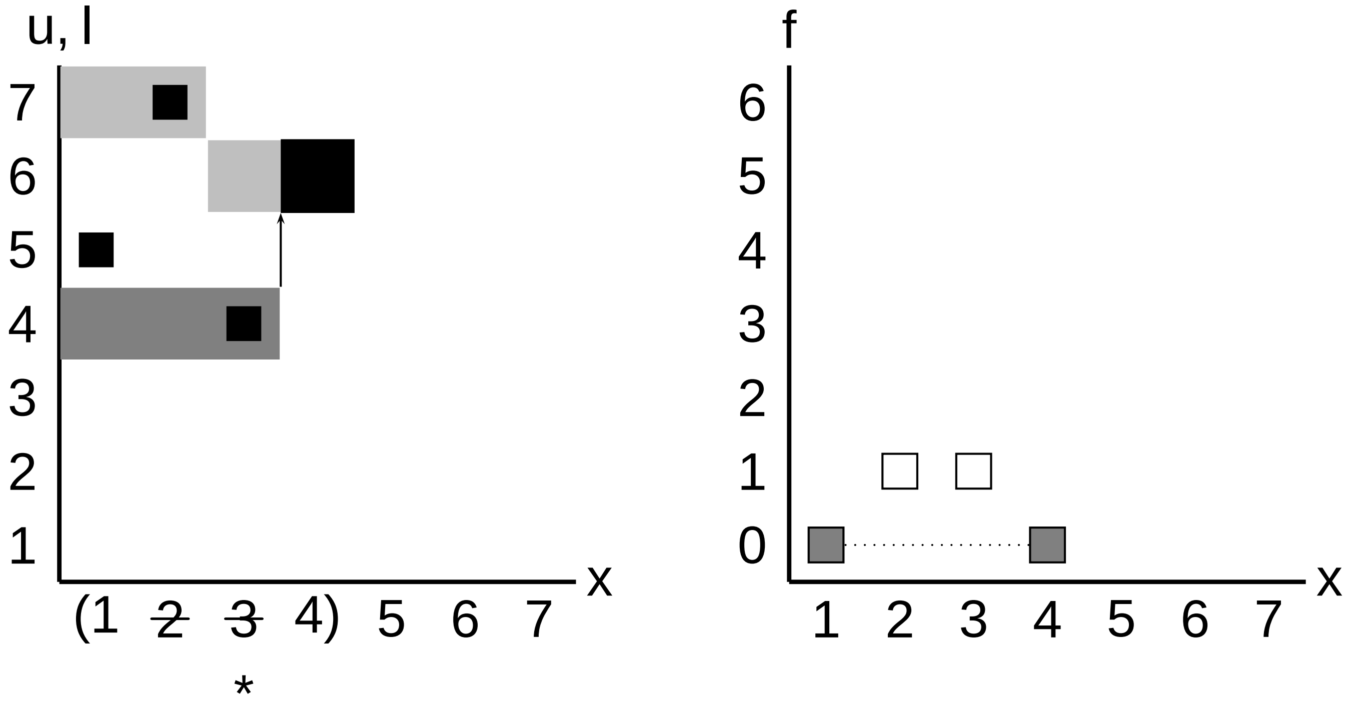


Candidate Elimination Elaborated



Example Execution Trace: A Snapshot

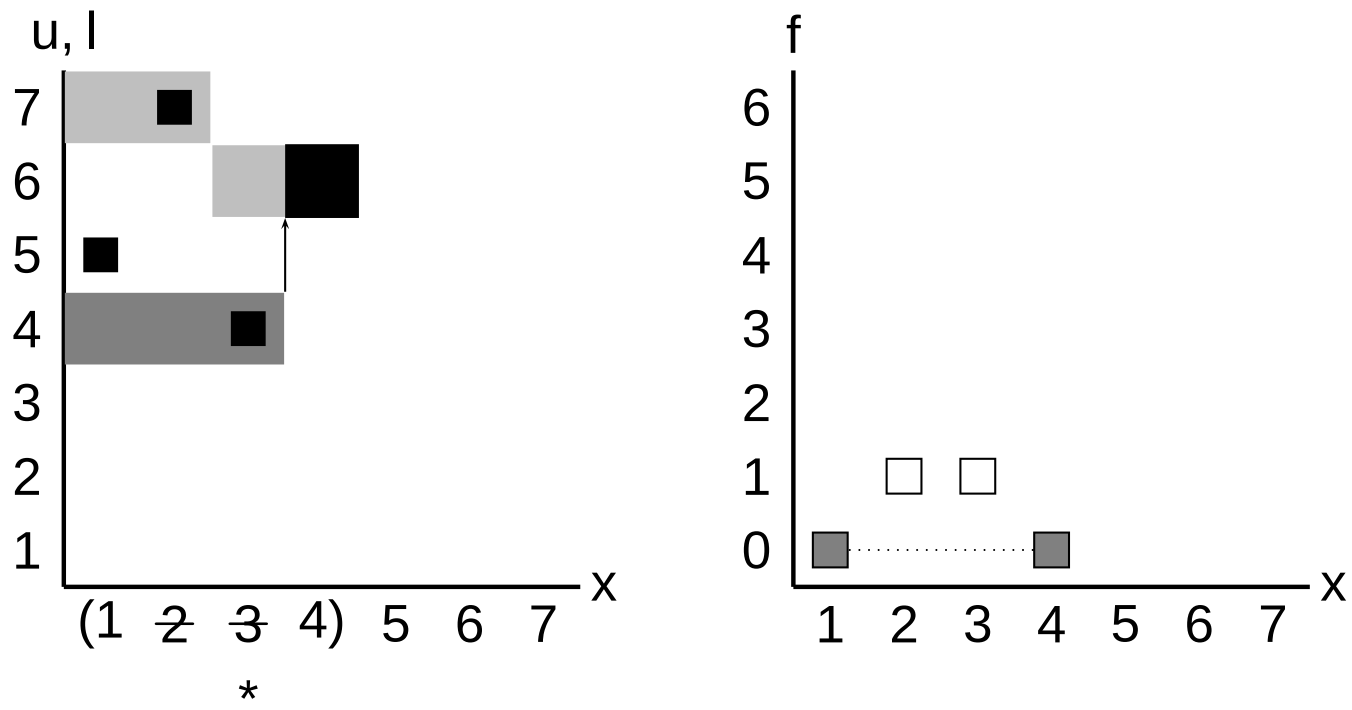
y = 4:



$$f(x, y) = u(x, y) - l(x, y) - (y - x)$$

Example Execution Trace: A Snapshot

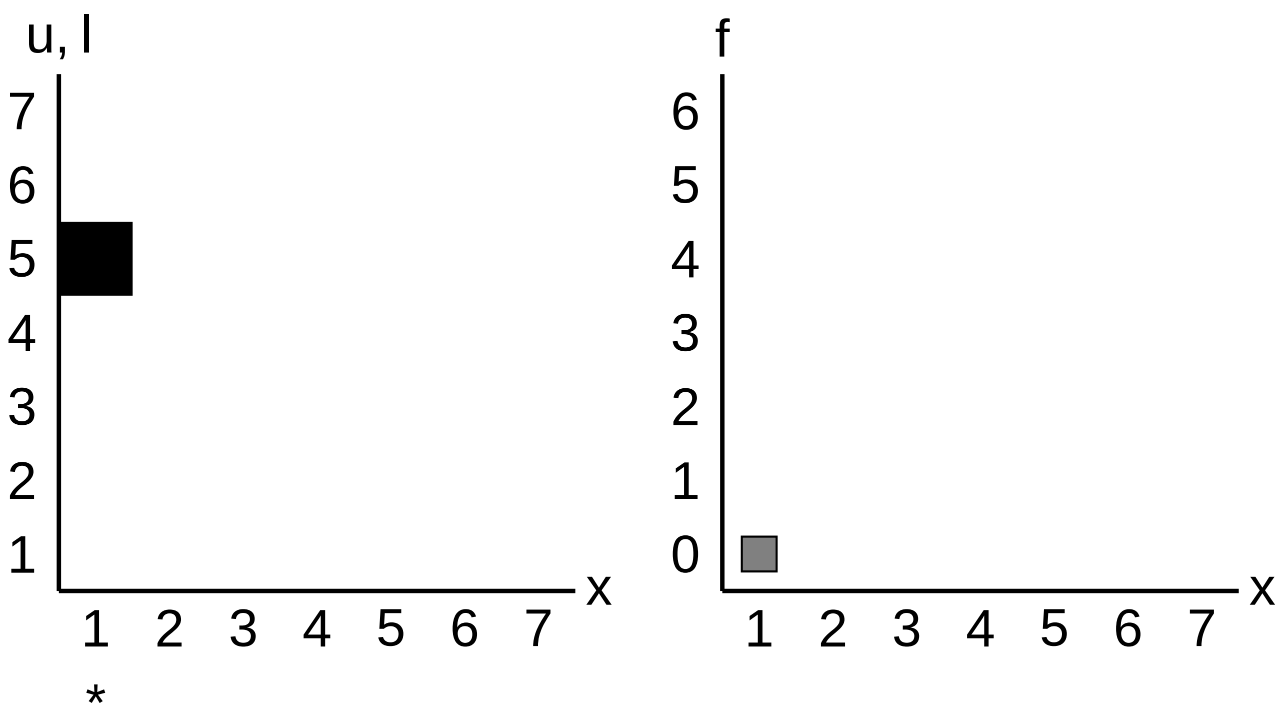
y = 4:



$$f(x, 4) = u(x, 4) - l(x, 4) - (4 - x)$$

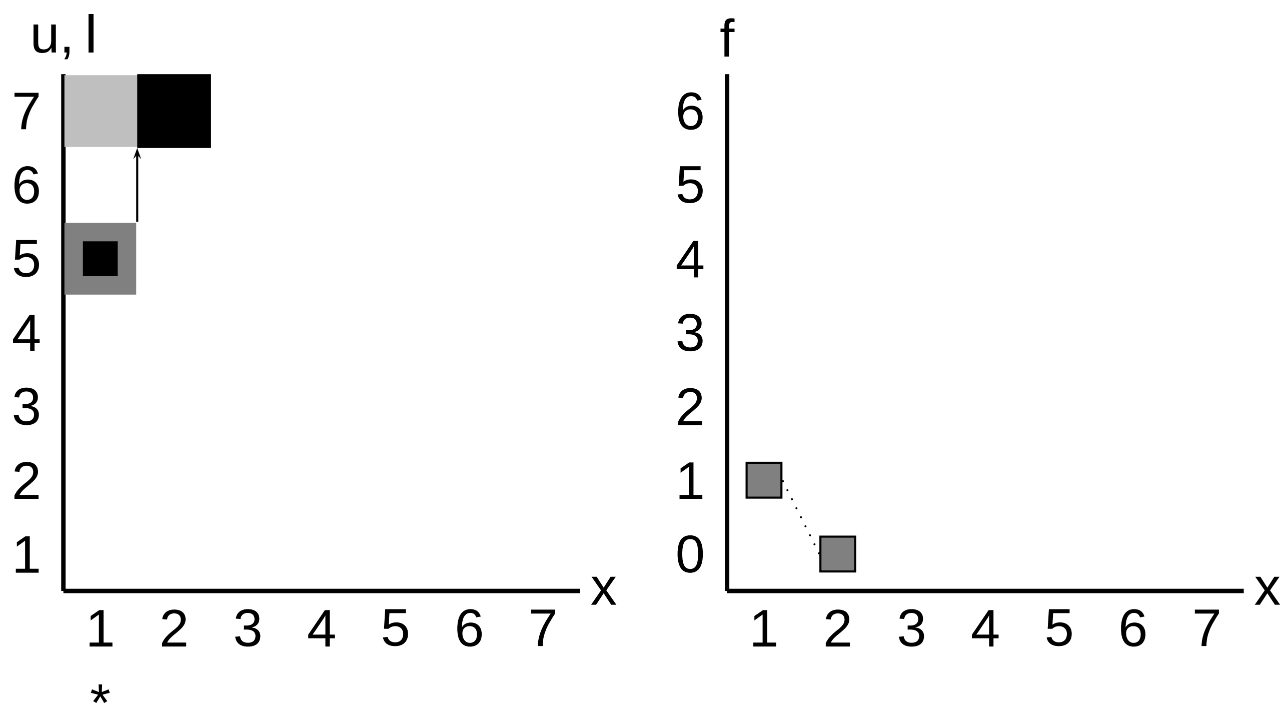
Example Execution Trace

y = 1:



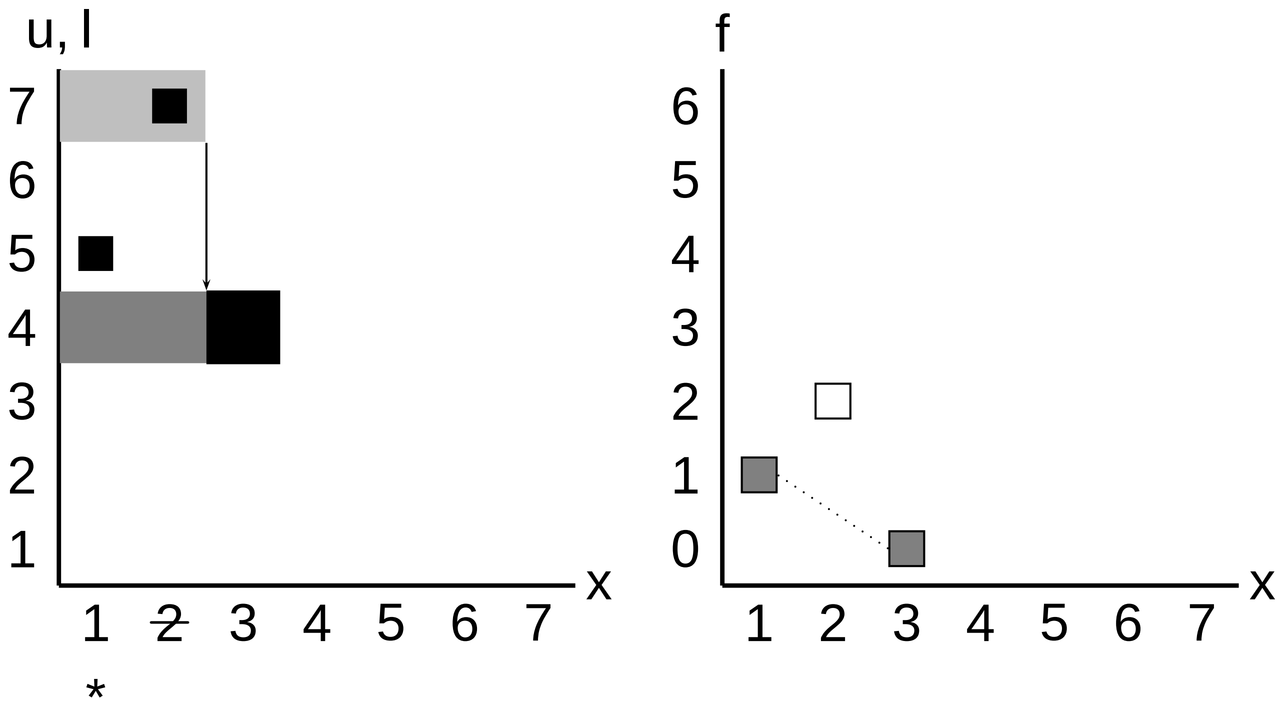
Example Execution Trace

y = 2:



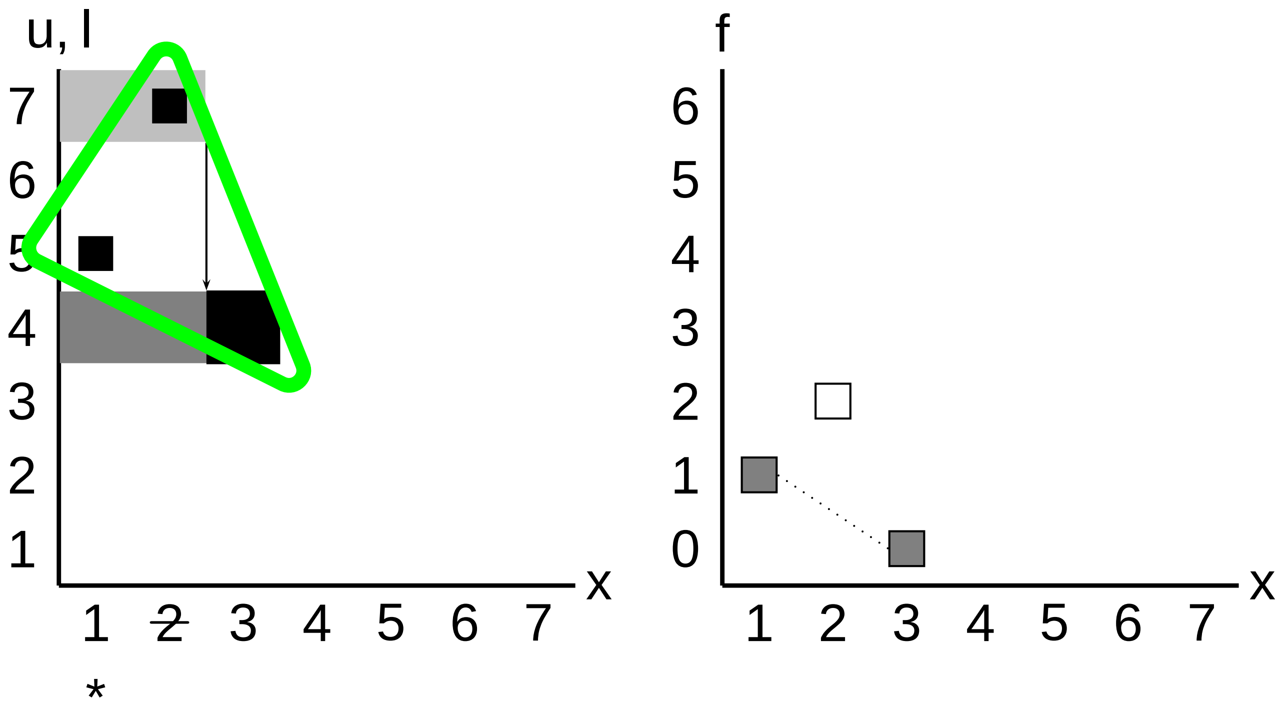
Example Execution Trace

y = 3:



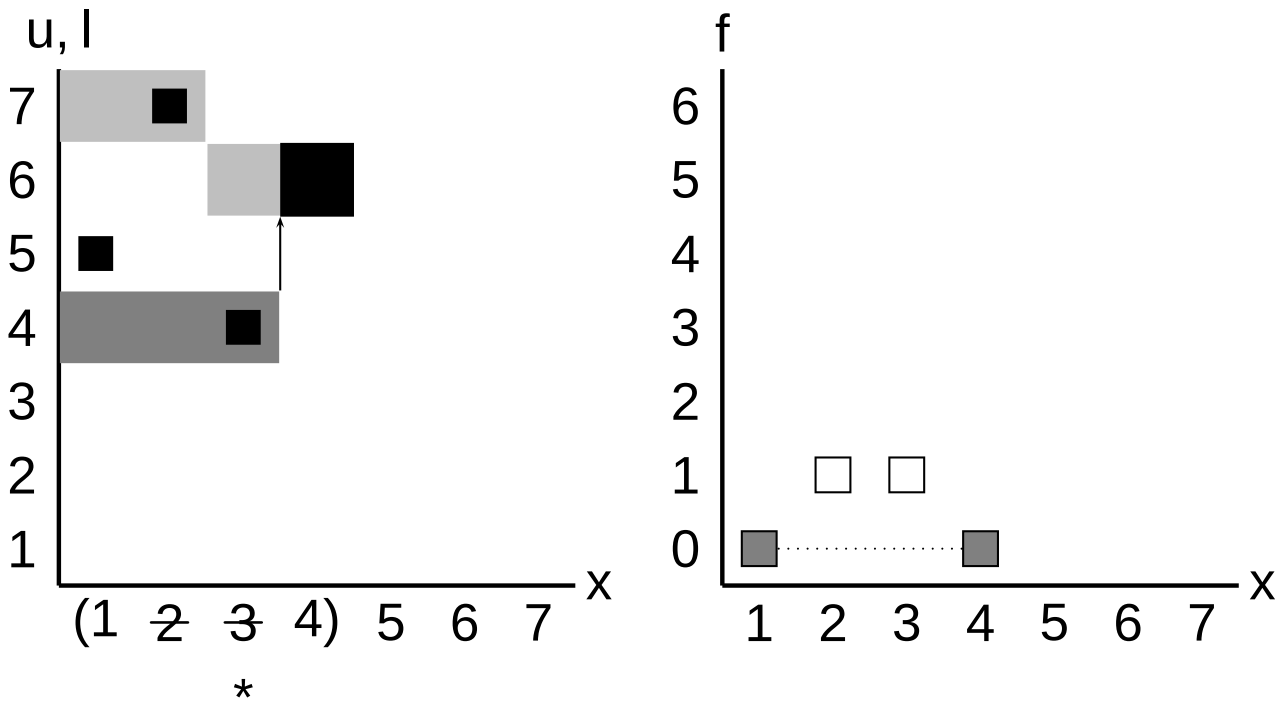
Example Execution Trace

y = 3:



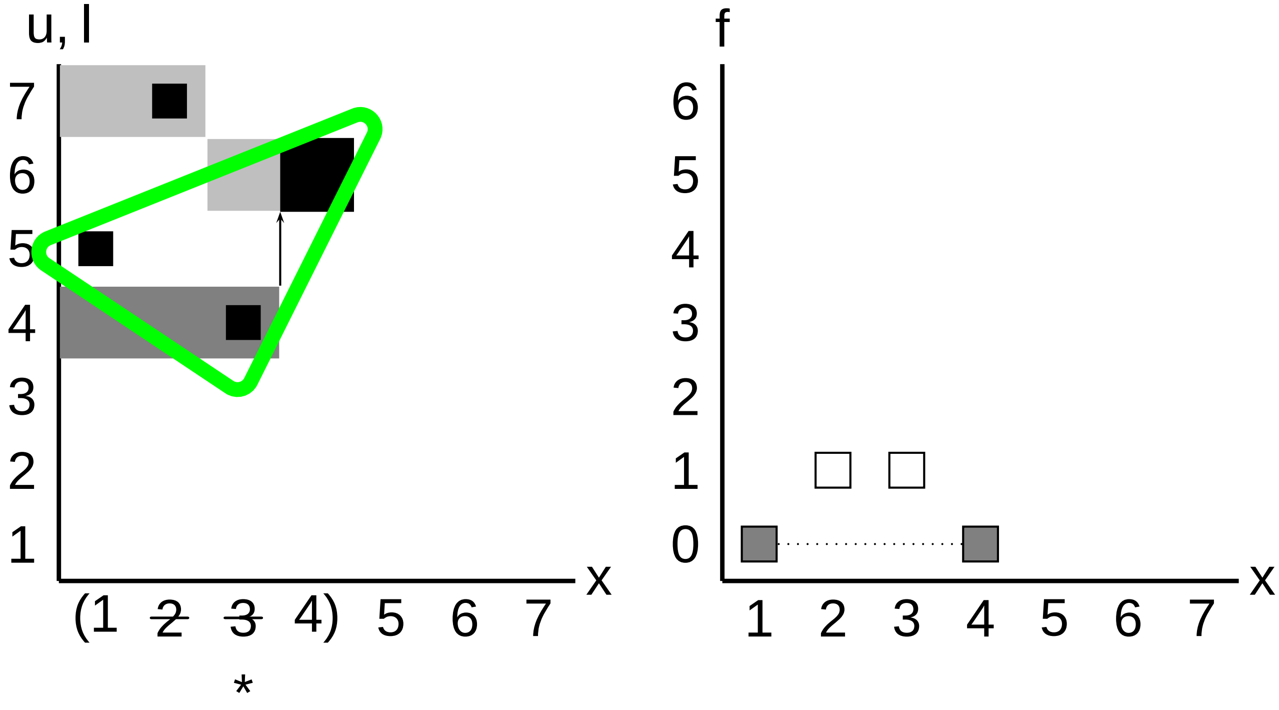
Example Execution Trace

y = 4:



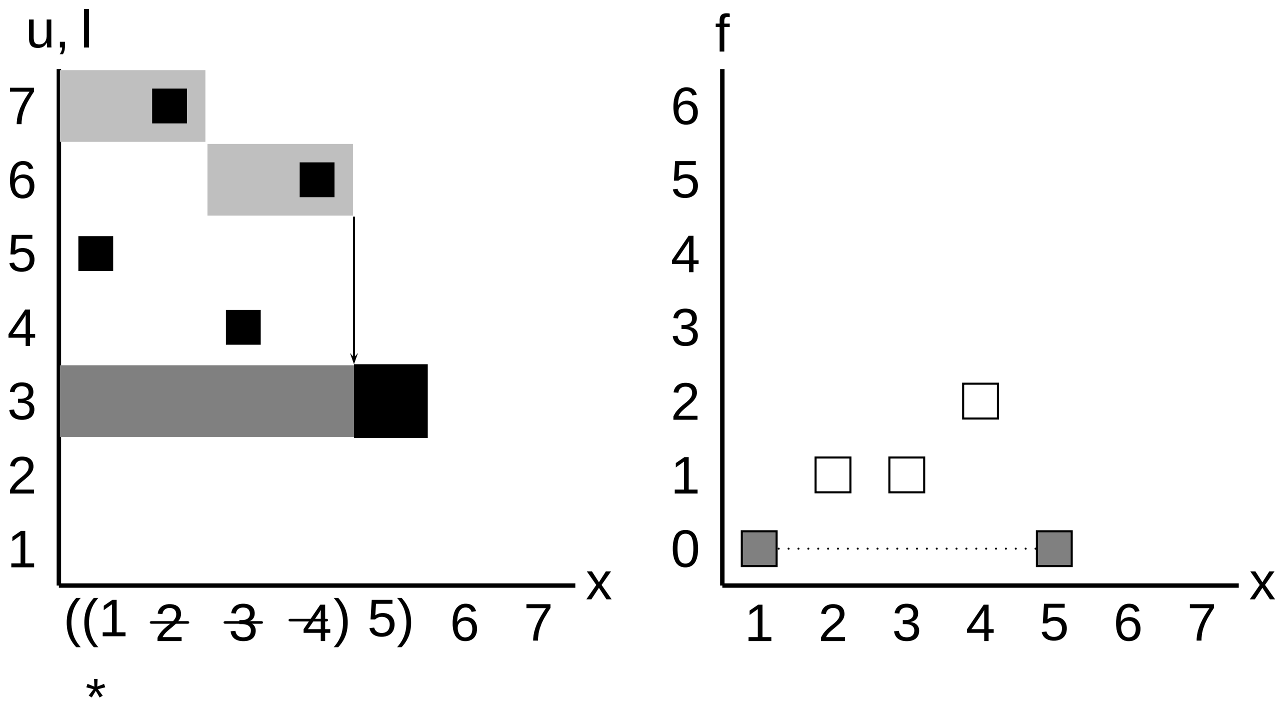
Example Execution Trace

y = 4:



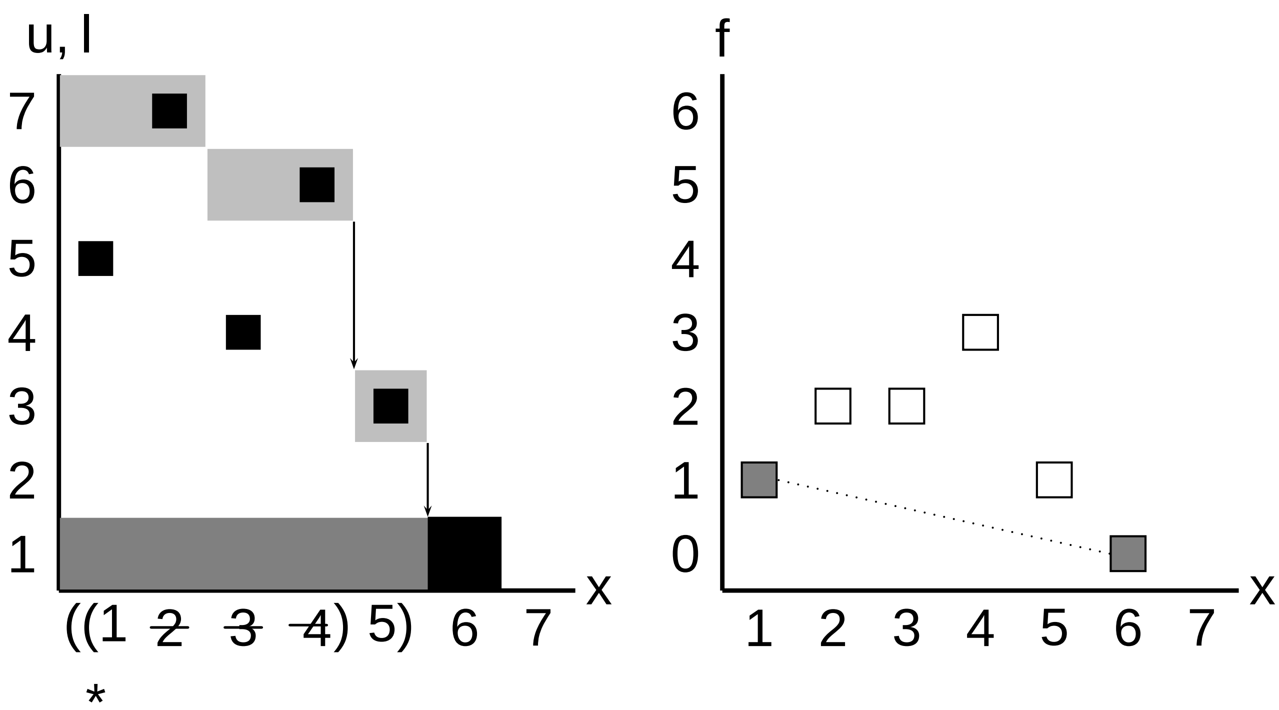
Example Execution Trace

y = 5:



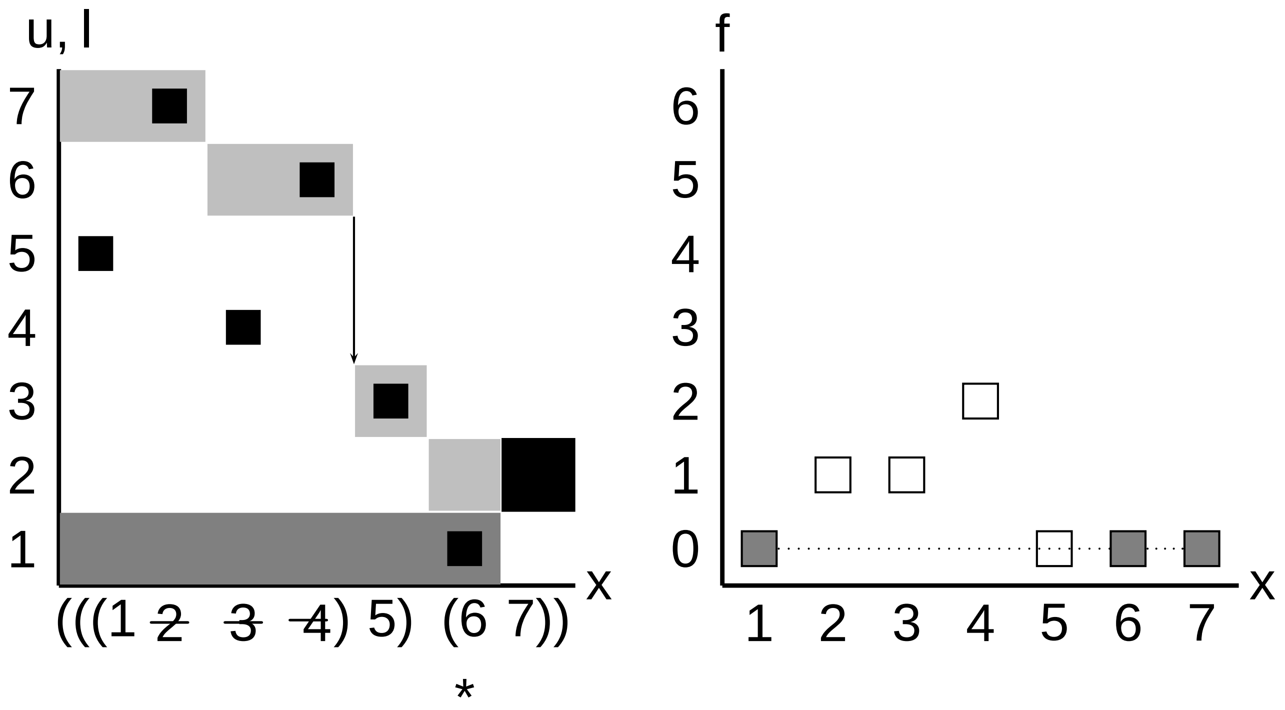
Example Execution Trace

y = 6:



Example Execution Trace

y = 7:



K-arizability of Permutations in Hand-aligned Data

	Branching Factor							
	1	2	4	5	6	7	10	$\geq$ 4 (and covering $>$ 10 words)
Chinese/English		451	30	4	5	1		7(1.4%)
Romanian/English		195	4					0
Hindi/English	3	85	1	1				0
Spanish/English		195	4					1(0.5%)
French/English		425	9	9	3		1	6(1.3%)

Conclusions

- A linear time algorithm exists for the SCFG factorization problem.
- The algorithm is truly efficient in the sense that it has a small constant factor.
- We analyze hand-aligned data sets for various language pairs, showing potentially the maximum branching factors needed for SCFGs for different language pairs.

Thanks