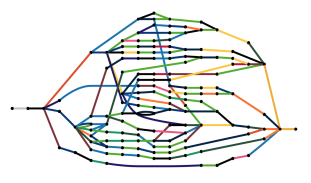


Inefficient Negative Selection

Efficient Negative Selection

Lecture 2

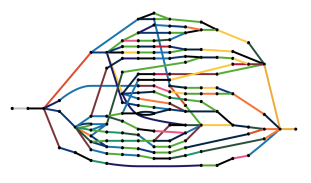
Efficient Algorithms for String-Based Negative Selection

Course Natural Computing, March 10th, 2020

Johannes Textor
Institute for Computing and Information Sciences

Learning Objectives 2

1. Understand why implementing a negative selection algorithm literally is almost always a bad idea.
2. Appreciate how string compression techniques can be used to drastically improve the performance of negative selection algorithms.



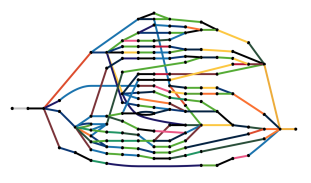
Inefficient Negative Selection

Efficient Negative Selection

Lecture Outline 2

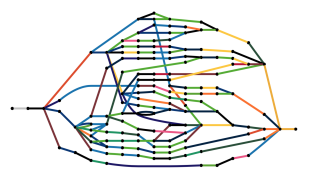
1. Inefficient Negative Selection

2. Efficient Negative Selection



Inefficient Negative Selection

Efficient Negative Selection



Inefficient Negative Selection

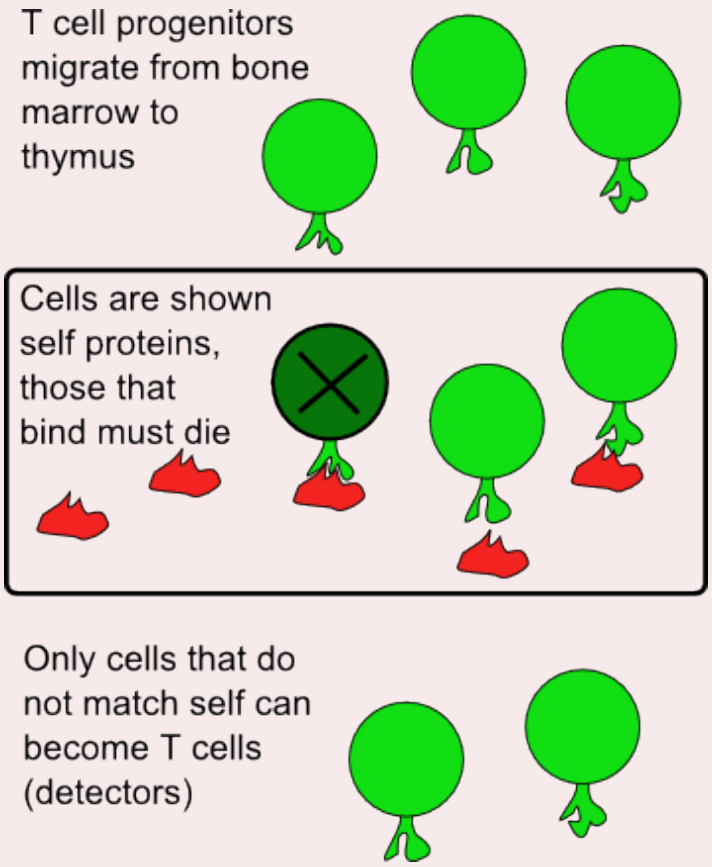
Efficient Negative Selection

"Of course it is very interesting to know how humans can learn. However, this is not necessarily the best way for creating an artificial learning machine. It has been noted that the study of birds flying was not very useful for constructing the airplane"

(Vladimir N. Vapnik, The Nature of Statistical Learning Theory)

General Structure of a Negative Selection Algorithm

Negative Selection in the IS

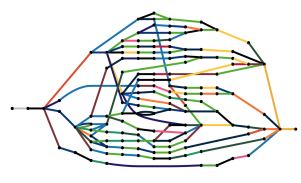


The negative selection algorithm

input: self-set $S \subseteq$ SELF,
monitoring set $M \subseteq \mathcal{X}$
output: For each $m \in M$:
either SELF or NON-SELF

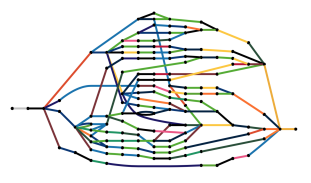
$D \leftarrow$ Set of detectors that do not match any $s \in S$
for each $m \in M$ **do**
  if m matches any $d \in D$ **then**
    output "m IS NON-SELF"
  else
    output "m IS SELF"

(Forrest, 1997)



Inefficient Negative Selection
Efficient Negative Selection

String-Based Negative Selection



Inefficient Negative Selection

Efficient Negative Selection

Consider the learning domain fixed-length binary strings $\mathcal{X} = \{0, 1\}^\ell$.

Self-set S

01000	11010
01001	11011
01010	
01011	

r-chunk detector

Pair of string s of length r and position i. Matches a string s' of length ℓ, if s occurs in s' at position i.

E.g. (010,3); (111,2)

r-contiguous detector

String s of length ℓ. Matches another string s', if s and s' contain the same letters at r subsequent positions.

E.g. 00011; 00100

String-Based Negative Selection

Consider the learning domain fixed-length binary strings $\mathcal{X} = \{0, 1\}^\ell$.

Self-set S

01000	11010
01001	11011
01010	
01011	

r-chunk detector

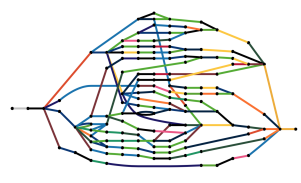
Pair of string s of length r and position i. Matches a string s' of length ℓ, if s occurs in s' at position i.

E.g. (010,3); (111,2)

r-contiguous detector

String s of length ℓ. Matches another string s', if s and s' contain the same letters at r subsequent positions.

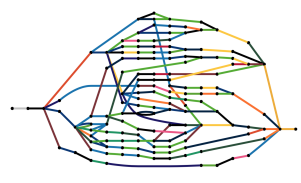
E.g. 00011; 00100



Inefficient Negative Selection

Efficient Negative Selection

String-Based Negative Selection



Inefficient Negative Selection

Efficient Negative Selection

Consider the learning domain fixed-length binary strings $\mathcal{X} = \{0, 1\}^\ell$.

Self-set S

01000	11010
01001	11011
01010	
01011	

r-chunk detector

Pair of string s of length r and position i. Matches a string s' of length ℓ, if s occurs in s' at position i.

E.g. (010,3); (111,2)

r-contiguous detector

String s of length ℓ. Matches another string s', if s and s' contain the same letters at r subsequent positions.

E.g. 00011; 00100

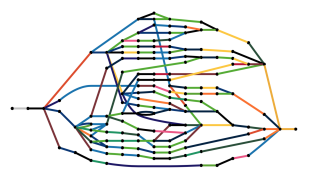
Generating Detectors Randomly

Self-Set S

```
1 1 0 0 0 1 0 1
0 0 1 1 0 0 1 0
0 0 1 1 1 1 0 0
0 0 1 0 0 1 0 0
1 0 1 1 1 0 1 0
```

Detector-Set D

```
1 1 0 1 1 0 1 1
1 1 1 0 1 1 1 1
0 1 0 0 1 0 0 1
0 1 1 1 0 0 0 1
```

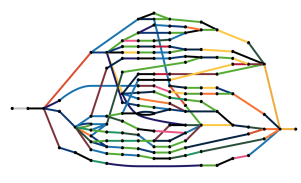



Inefficient Negative Selection

Efficient Negative Selection

Our Goal: Infer Detectors Directly

- Generating detectors randomly is common in practice *(Hofmeyr '99, Balthrop '02, and many others)*

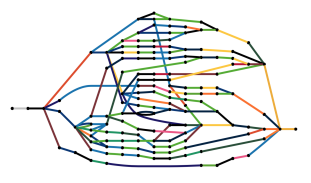


Inefficient Negative Selection

Efficient Negative Selection

Our Goal: Infer Detectors Directly

- Generating detectors randomly is common in practice *(Hofmeyr '99, Balthrop '02, and many others)*
- Efficiency issues:
  - Too many detctors generable - generating all is too slow
  - Few or no detectors generable - finding them is too hard

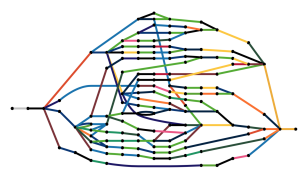


Inefficient Negative Selection

Efficient Negative Selection

Our Goal: Infer Detectors Directly

- Generating detectors randomly is common in practice *(Hofmeyr '99, Balthrop '02, and many others)*
- Efficiency issues:
  - Too many detctors generable - generating all is too slow
  - Few or no detectors generable - finding them is too hard
- Can't we infer detectors directly?
  - Counting r-contiguous detectors is feasible in time polynomial in L and $|S|$, but exponential in r *(D'Haeseleer '96)*
  - Bounds for $|S|$ with respect to detector generability *(Stibor '08)*
  - Reduction of detector generability to r-SAT *(Stibor '06)*

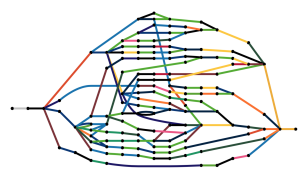


Inefficient Negative Selection

Efficient Negative Selection

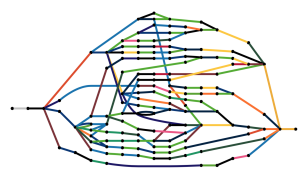
Our Goal: Infer Detectors Directly

- Generating detectors randomly is common in practice *(Hofmeyr '99, Balthrop '02, and many others)*
- Efficiency issues:
  - Too many detctors generable - generating all is too slow
  - Few or no detectors generable - finding them is too hard
- Can't we infer detectors directly?
  - Counting r-contiguous detectors is feasible in time polynomial in L and $|S|$, but exponential in r *(D'Haeseleer '96)*
  - Bounds for $|S|$ with respect to detector generability *(Stibor '08)*
  - Reduction of detector generability to r-SAT *(Stibor '06)*
- Our result: string-based negative selection is feasible in polynomial time!



Inefficient Negative Selection

Efficient Negative Selection



Inefficient Negative Selection

Efficient Negative Selection

Negative selection [...] turns out to be equivalent to an NP-complete problem.

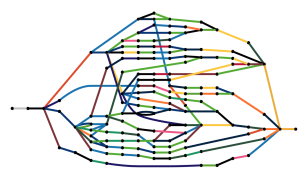
(Timmis et al, 2008)

"Negative selection was thoroughly explored and has no potential for becoming a [...] useful [...] technique. [...] Future work [...] is not meaningful."

(A dissertation, 2006)

The Key Trick: Compressing Complement Sets

Problem: Complement of a string set S is much larger than S itself.
Solution: Compress the complement using a “don’t care symbol”.



Inefficient Negative Selection

Efficient Negative Selection

The Key Trick: Compressing Complement Sets

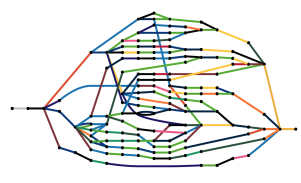
Problem: Complement of a string set S is much larger than S itself.
Solution: Compress the complement using a "don't care symbol".

Minimal prefix patterns *(Esponda '05)*

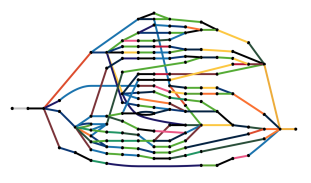
Input: Set $S \subseteq \mathcal{U}$
Output: Set of patterns describing the complement of S

for each prefix π of any $s \in S$ **do**
  flip last bit of π
  unless π is now a prefix
    of another $s \in S$ **do**
      pad π to the right with $*$
      output π

Time: $O(|S|r^2)$. Elements: $O(|S|r)$



Inefficient Negative Selection
Efficient Negative Selection



The Key Trick: Compressing Complement Sets

Problem: Complement of a string set S is much larger than S itself.
Solution: Compress the complement using a "don't care symbol".

String set S

01000	11010
01001	11011
01010	
01011	

$\textsc{MinPrefix}(\bar{S})$

Minimal prefix patterns *(Esponda '05)*

Input: Set $S \subseteq \mathcal{U}$
Output: Set of patterns describing the complement of S

for each prefix π of any $s \in S$ **do**
  flip last bit of π
  unless π is now a prefix
    of another $s \in S$ **do**
      pad π to the right with $*$
      output π

Time: $O(|S|r^2)$. Elements: $O(|S|r)$

The Key Trick: Compressing Complement Sets

Problem: Complement of a string set S is much larger than S itself.
Solution: Compress the complement using a "don't care symbol".

String set S

01000	11010
01001	11011
01010	
01011	

MinPrefix($\bar{S}$)

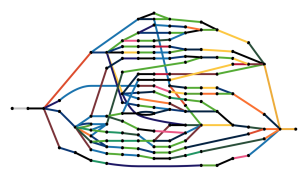
00***

Minimal prefix patterns *(Esponda '05)*

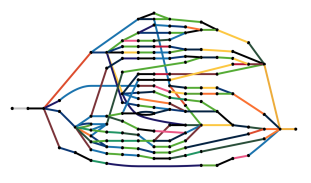
Input: Set $S \subseteq \mathcal{U}$
Output: Set of patterns describing the complement of S

for each prefix π of any $s \in S$ **do**
  flip last bit of π
  unless π is now a prefix
    of another $s \in S$ **do**
      pad π to the right with $*$
      output π

Time: $O(|S|r^2)$. Elements: $O(|S|r)$



Inefficient Negative Selection
Efficient Negative Selection



The Key Trick: Compressing Complement Sets

Problem: Complement of a string set S is much larger than S itself.
Solution: Compress the complement using a "don't care symbol".

String set S

01000 11010
01001 11011
01010
01011

MinPrefix($\bar{S}$)

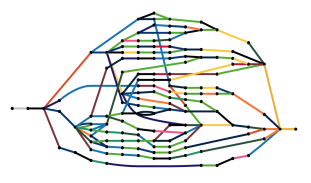
00***
011**

Minimal prefix patterns *(Esponda '05)*

Input: Set $S \subseteq \mathcal{U}$
Output: Set of patterns describing the complement of S

for each prefix π of any $s \in S$ **do**
  flip last bit of π
  unless π is now a prefix of another $s \in S$ **do**
    pad π to the right with $*$
    output π

Time: $O(|S|r^2)$. Elements: $O(|S|r)$



The Key Trick: Compressing Complement Sets

Problem: Complement of a string set S is much larger than S itself.
Solution: Compress the complement using a "don't care symbol".

String set S

01000 11010
01001 11011
01010
01011

MinPrefix($\bar{S}$)

00*** 10***
011**

Minimal prefix patterns *(Esponda '05)*

Input: Set $S \subseteq \mathcal{U}$
Output: Set of patterns describing the complement of S

for each prefix π of any $s \in S$ **do**
  flip last bit of π
  unless π is now a prefix of another $s \in S$ **do**
    pad π to the right with $*$
    output π

Time: $O(|S|r^2)$. Elements: $O(|S|r)$

The Key Trick: Compressing Complement Sets

Problem: Complement of a string set S is much larger than S itself.
Solution: Compress the complement using a "don't care symbol".

String set S

01000	11010
01001	11011
01010	
01011	

MinPrefix($\bar{S}$)

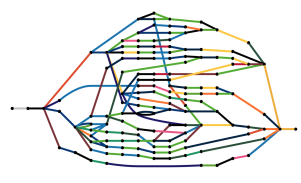
00***	10***
011**	111**

Minimal prefix patterns *(Esponda '05)*

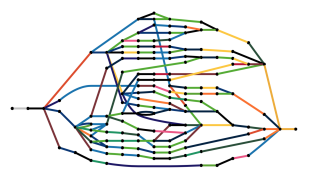
Input: Set $S \subseteq \mathcal{U}$
Output: Set of patterns describing the complement of S

for each prefix π of any $s \in S$ **do**
  flip last bit of π
  unless π is now a prefix
    of another $s \in S$ **do**
      pad π to the right with $*$
      output π

Time: $O(|S|r^2)$. Elements: $O(|S|r)$



Inefficient Negative Selection
Efficient Negative Selection



Inefficient Negative Selection
Efficient Negative Selection

The Key Trick: Compressing Complement Sets

Problem: Complement of a string set S is much larger than S itself.
Solution: Compress the complement using a "don't care symbol".

String set S

01000	11010
01001	11011
01010	
01011	

MinPrefix($\bar{S}$)

00***	10***
011**	111**
	1100*

Minimal prefix patterns *(Esponda '05)*

Input: Set $S \subseteq \mathcal{U}$
Output: Set of patterns describing the complement of S

for each prefix π of any $s \in S$ **do**
  flip last bit of π
  unless π is now a prefix of another $s \in S$ **do**
    pad π to the right with $*$
    output π

Time: $O(|S|r^2)$. Elements: $O(|S|r)$

r-Chunk Detection With Prefix Patterns

- r-chunk detectors are complement sets for $S[i, i+r]$
- We can use prefix patterns as detectors
- This results in at most $O(|S|Lr)$ detectors

Example: $|S| = 3, L = 5, r = 3$

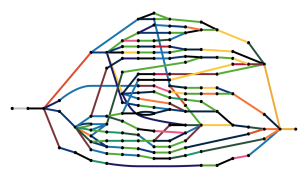
self-set S

```
0 1 0 1 1
0 1 0 1 0
0 1 1 0 1
```

detector set D

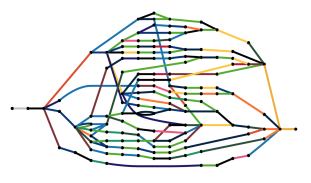
```
0 0 * * *        * 0 * * *        * * 0 0 *

1 * * * *                         * * 1 0 0
                                  * * 1 1 *
                 * 1 0 0 *
                 * 1 1 1 *
```

Inefficient Negative Selection

Efficient Negative Selection

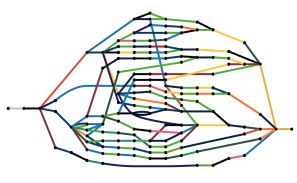


The *r*-Contiguous Graph

- *r*-contiguous detectors correspond to overlapping *r*-chunks
- Arrange overlapping *r*-chunk detectors in a graph. Then *r*-cont. detectors are paths on this graph!
- Can the graph be compressed?
- Can we construct the compressed graph directly?

Overlapping *r*-chunks

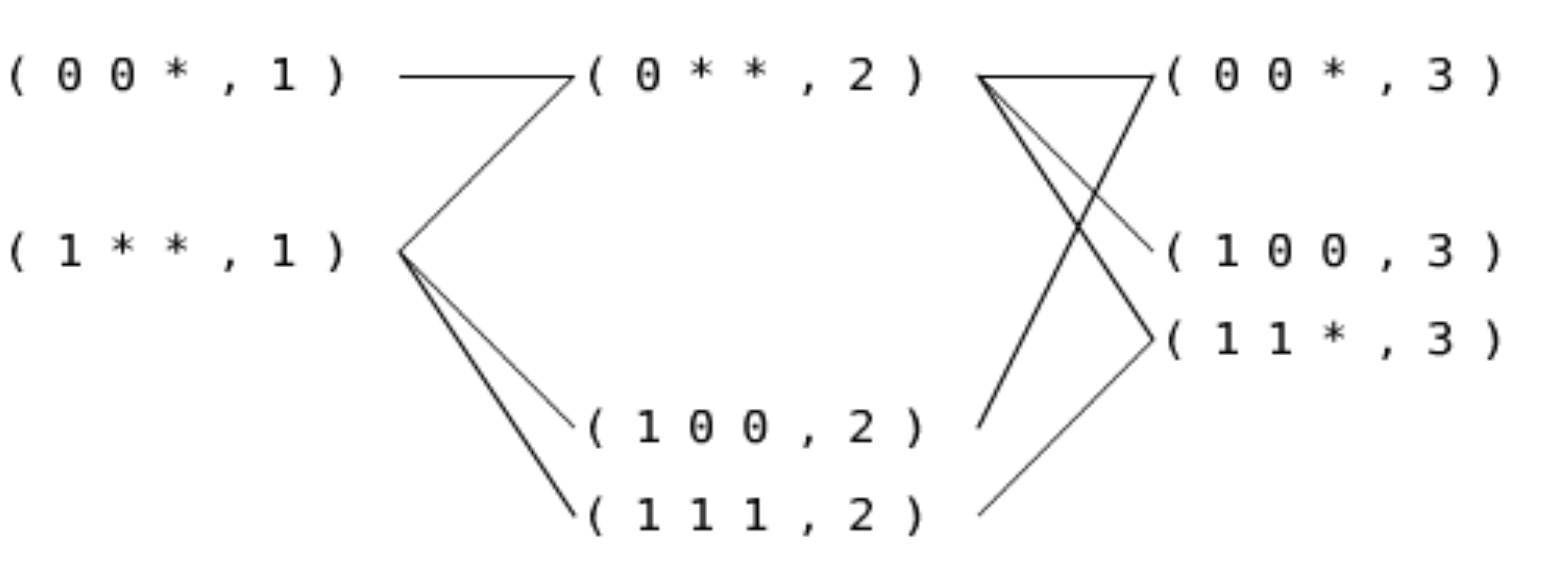
```
  0 0 0 1 0
( 0 0 0 , 1 )
  ( 0 0 1 , 2 )
    ( 0 1 0 , 3 )
```



The *r*-Contiguous Graph

- *r*-contiguous detectors correspond to overlapping *r*-chunks
- Arrange overlapping *r*-chunk detectors in a graph. Then *r*-cont. detectors are paths on this graph!
- Can the graph be compressed?
- Can we construct the compressed graph directly?

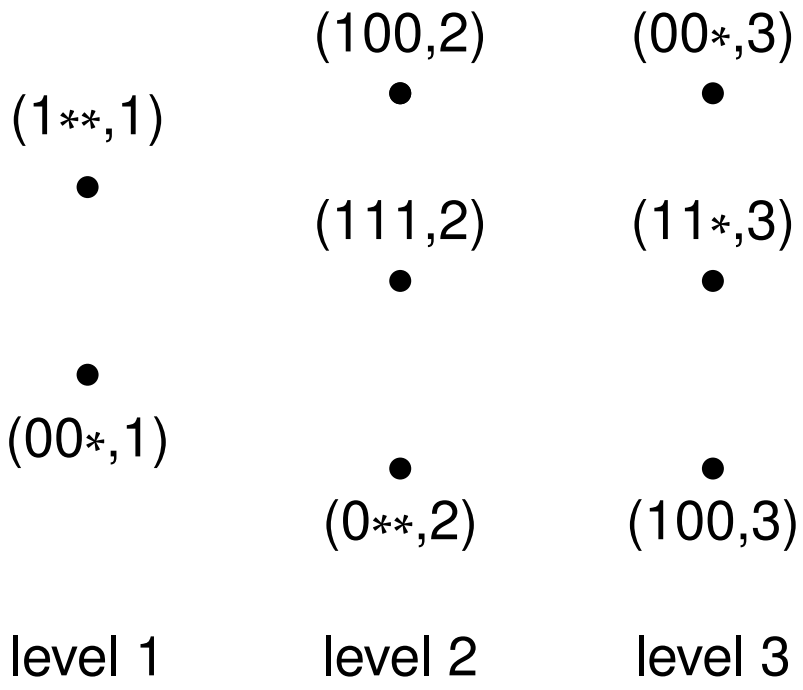
The *r*-contiguous graph

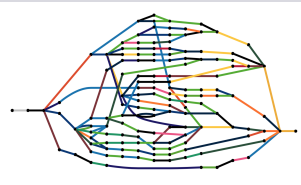


Constructing the r-Pattern Graph

Construction algorithm

1. Generate r-chunk patterns, arrange in levels
2. Link vertex (π, i) to vertex $(\pi', i+1)$ if $P(\pi[2,\ldots,r]) \supseteq P(\pi'[1,\ldots,r-1])$.
   e.g. ,
3. Drop vertices without incoming and outgoing edges.



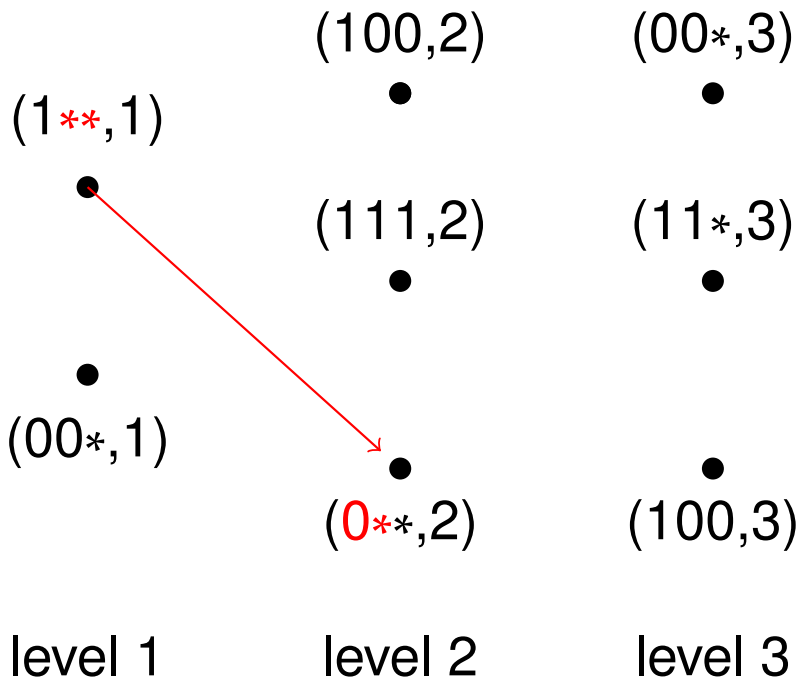


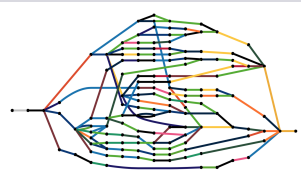
Inefficient Negative Selection
Efficient Negative Selection

Constructing the r-Pattern Graph

Construction algorithm

1. Generate r-chunk patterns, arrange in levels
2. Link vertex (π, i) to vertex $(\pi', i+1)$ if $P(\pi[2,\ldots,r]) \supseteq P(\pi'[1,\ldots,r-1])$.
   e.g. $P(**) \supseteq P(0*)$,
3. Drop vertices without incoming and outgoing edges.



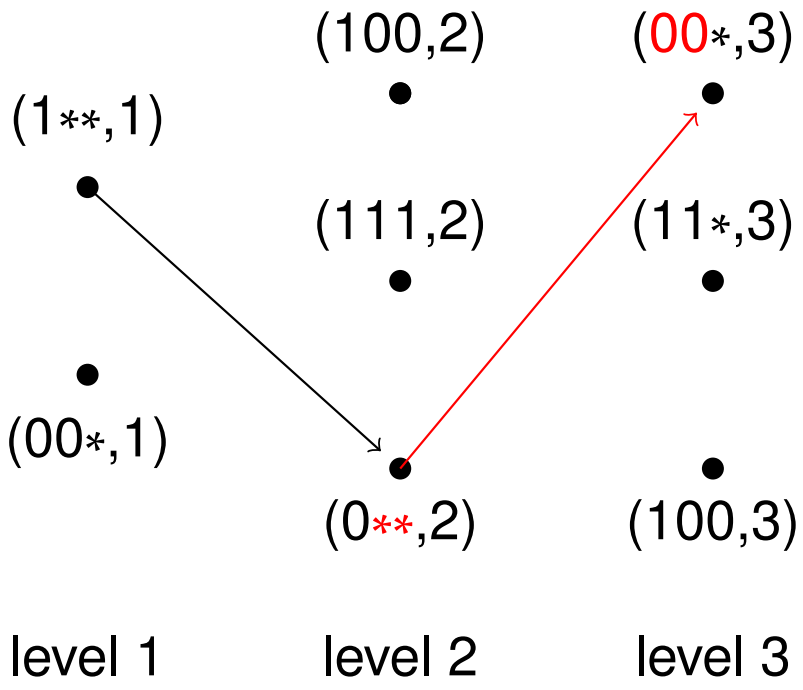


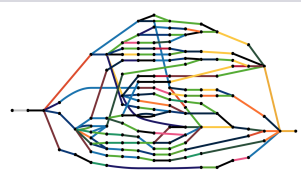
Inefficient Negative Selection
Efficient Negative Selection

Constructing the r-Pattern Graph

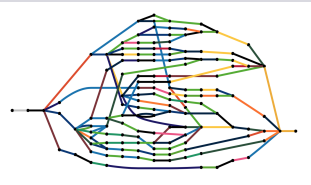
Construction algorithm

1. Generate r-chunk patterns, arrange in levels
2. Link vertex (π, i) to vertex $(\pi', i+1)$ if $P(\pi[2,\ldots,r]) \supseteq P(\pi'[1,\ldots,r-1])$.
   e.g. $P(**) \supseteq P(0*)$, $P(0*) \supseteq P(00)$
3. Drop vertices without incoming and outgoing edges.





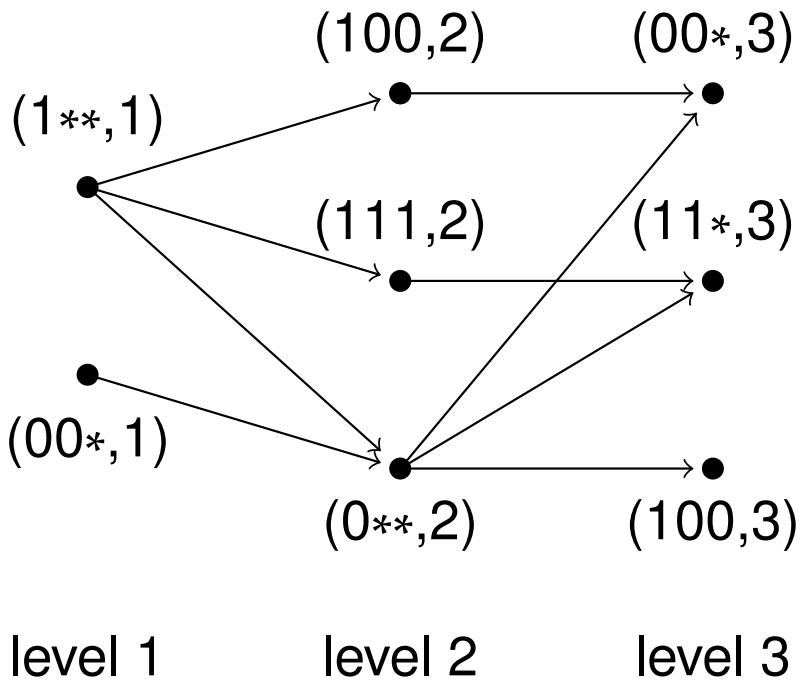
Inefficient Negative Selection
Efficient Negative Selection



Constructing the r-Pattern Graph

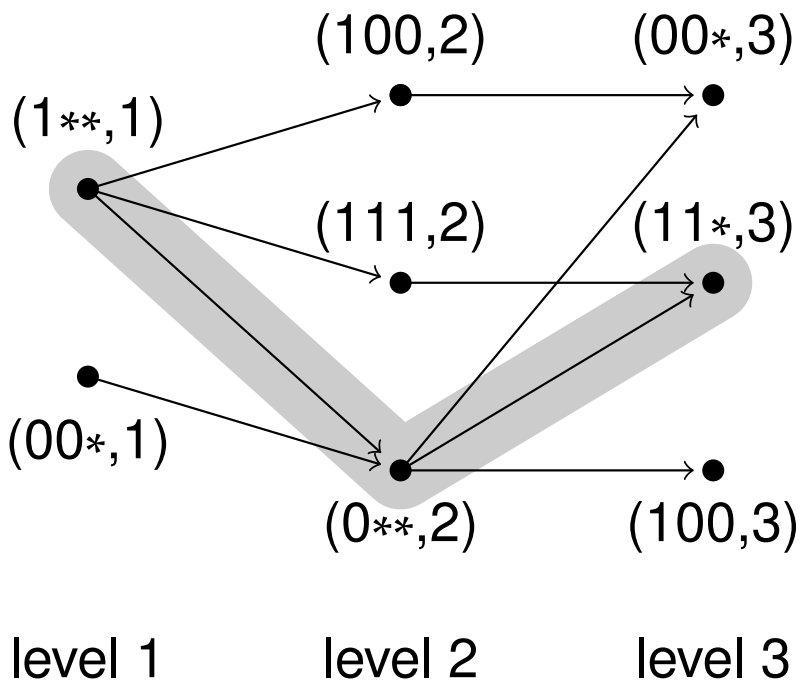
Construction algorithm

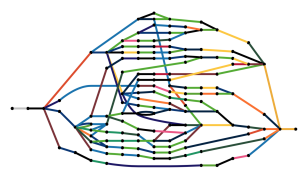
1. Generate r-chunk patterns, arrange in levels
2. Link vertex (π, i) to vertex $(\pi', i+1)$ if $P(\pi[2,\ldots,r]) \supseteq P(\pi'[1,\ldots,r-1])$.
   e.g. $P(**) \supseteq P(0*)$, $P(0*) \supseteq P(00)$
3. Drop vertices without incoming and outgoing edges.



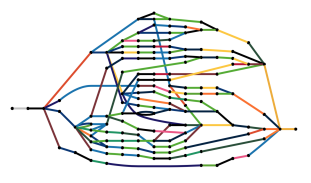
Correctness of the *r*-Pattern Graph

1. Every path in the *r*-Pattern Graph describes a set of *r*-contiguous detectors.
2. For every *r*-contiguous detector *d*, the *r*-Pattern Graph contains exactly one path that describes *d*.



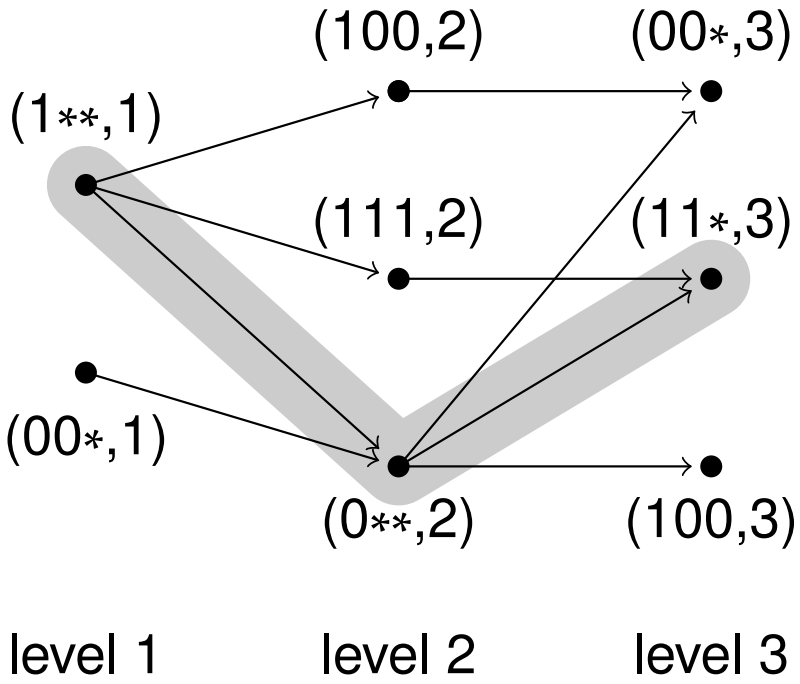


Inefficient Negative Selection
Efficient Negative Selection



Correctness of the *r*-Pattern Graph

1. Every path in the *r*-Pattern Graph describes a set of *r*-contiguous detectors.
2. For every *r*-contiguous detector *d*, the *r*-Pattern Graph contains exactly one path that describes *d*.

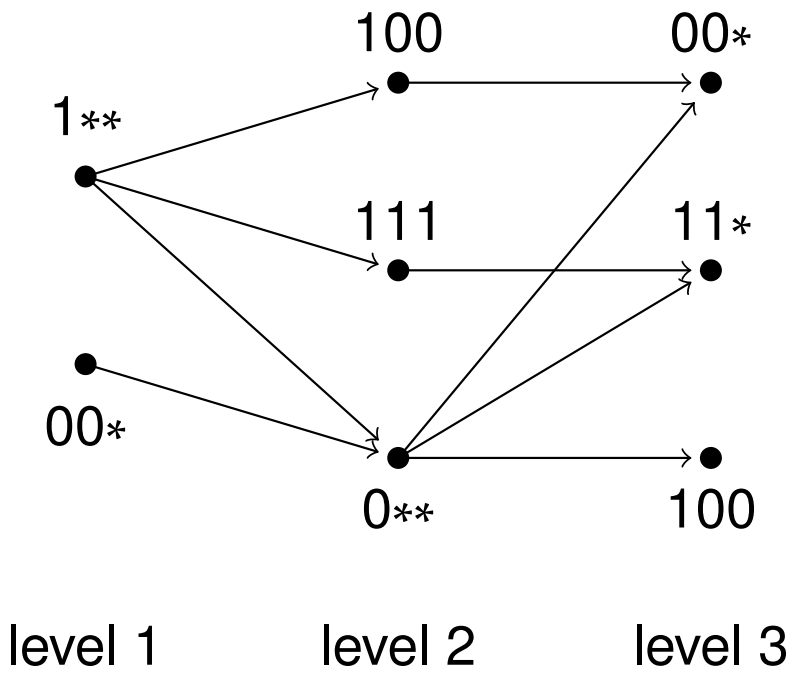


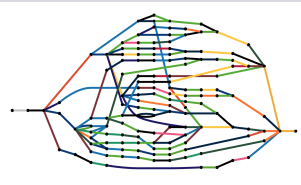
Merging patterns:

1	∗	∗		
	0	∗	∗	
		1	1	∗
1	0	1	1	0
1	0	1	1	1

Detection With the r-Pattern Graph

Transform r-pattern graph for detection

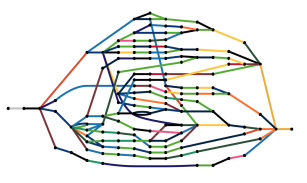




Inefficient Negative Selection

Efficient Negative Selection

Detection With the r-Pattern Graph

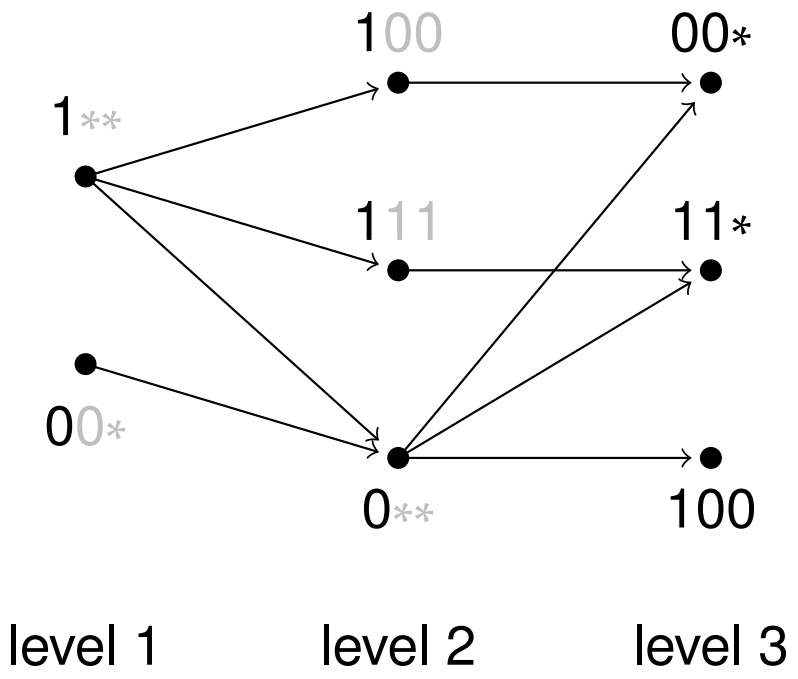


Inefficient Negative Selection

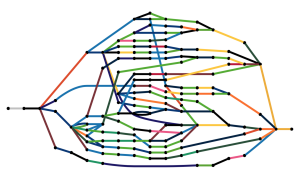
Efficient Negative Selection

Transform r-pattern graph for detection

1. Drop tail from all patterns on all levels except last



Detection With the r-Pattern Graph

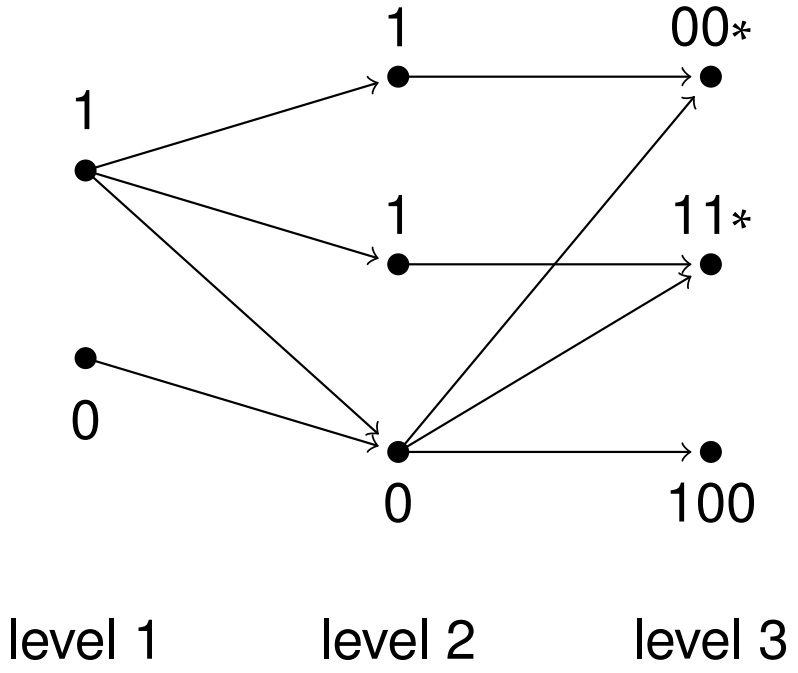


Inefficient Negative Selection

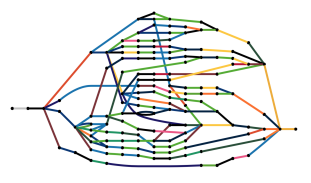
Efficient Negative Selection

Transform r-pattern graph for detection

1. Drop tail from all patterns on all levels except last



Detection With the r-Pattern Graph

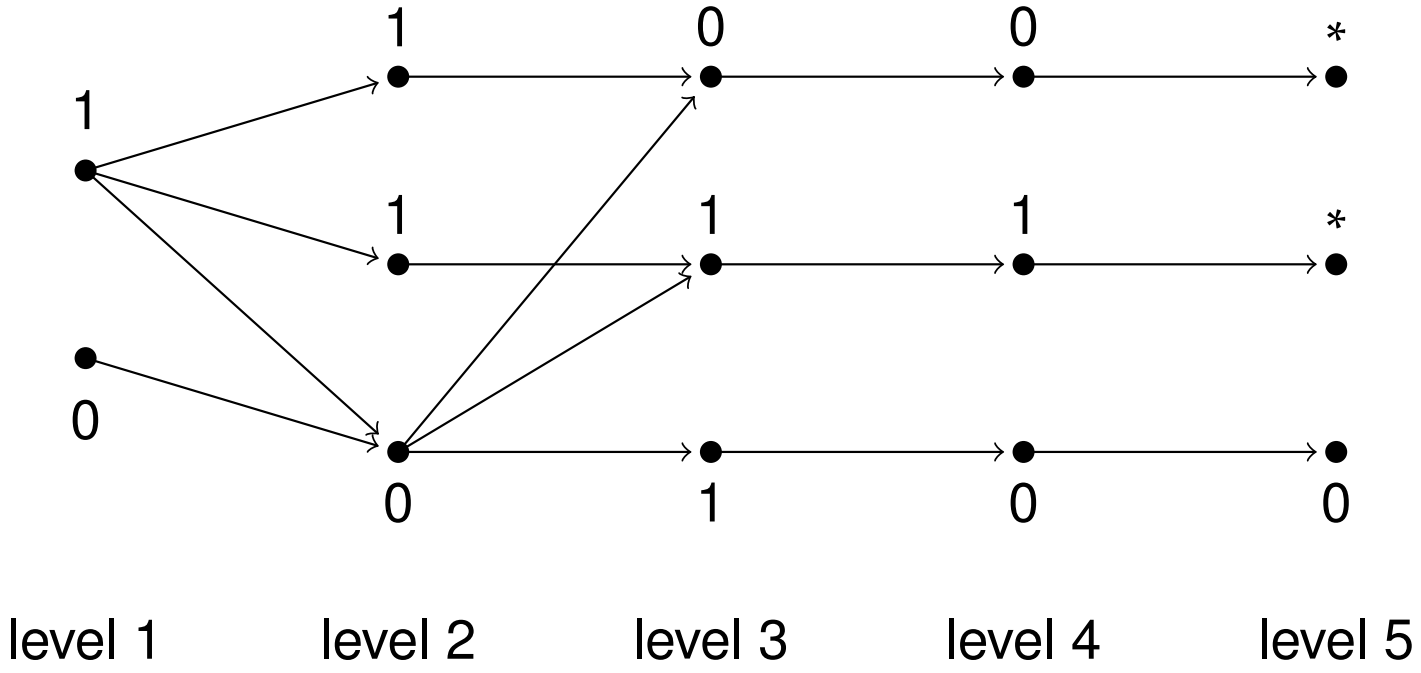


Inefficient Negative Selection

Efficient Negative Selection

Transform r-pattern graph for detection

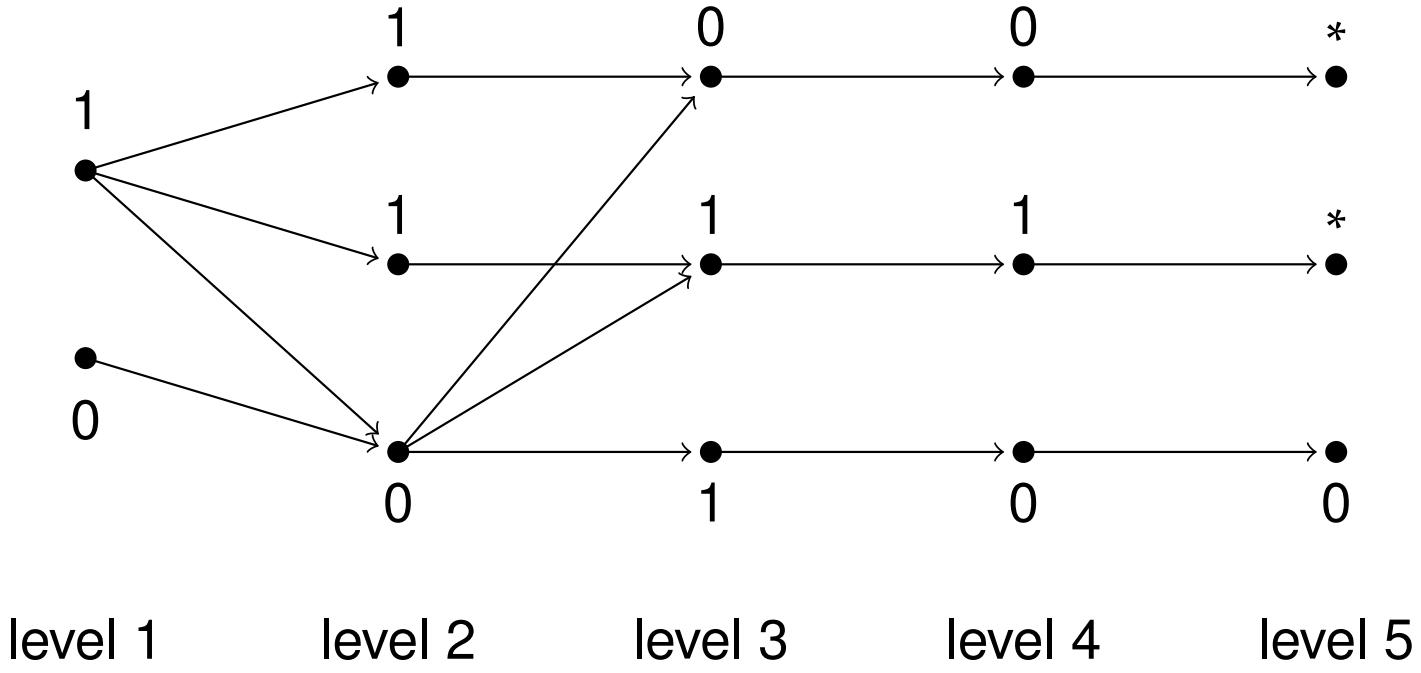
1. Drop tail from all patterns on all levels except last
2. Expand last level, creating $r - 1$ new levels

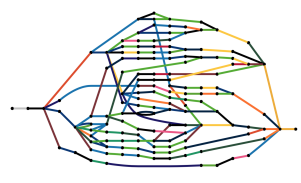


Detection With the *r*-Pattern Graph

Detection with the transformed graph

1. Depth first search for paths matching the input string





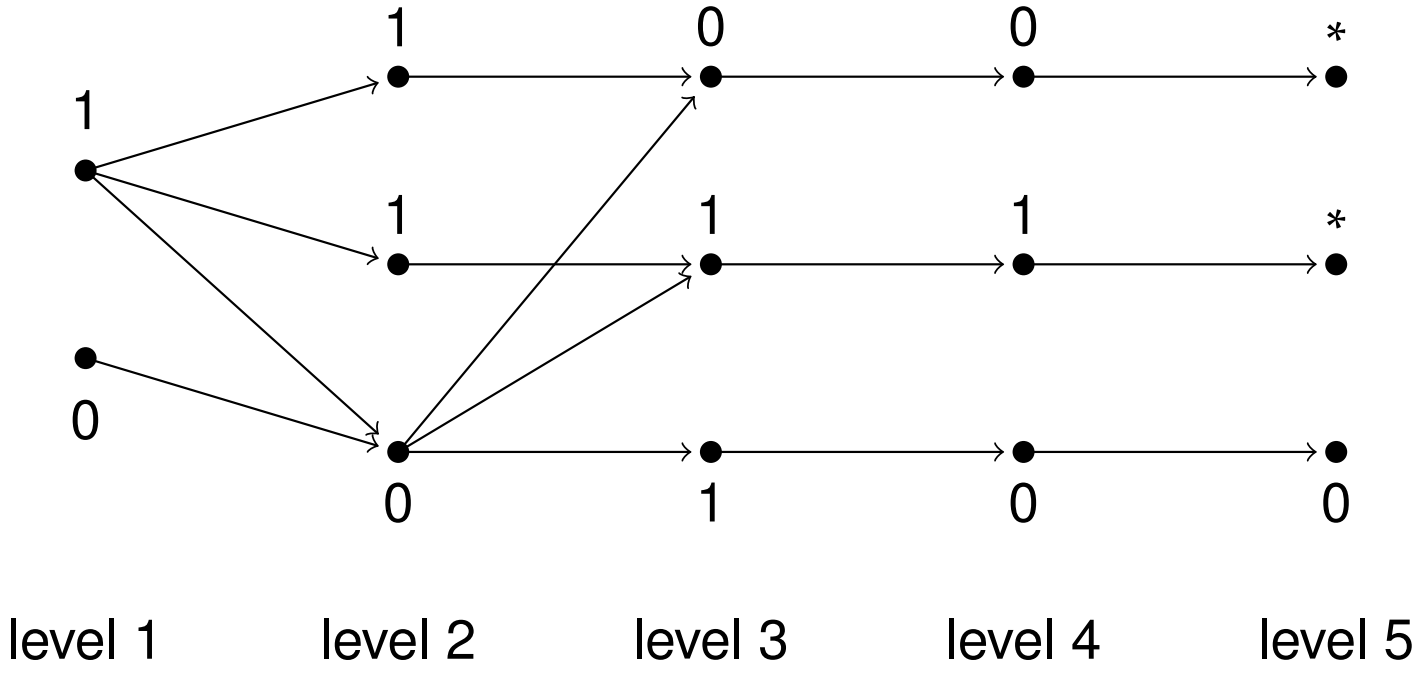
Inefficient Negative Selection

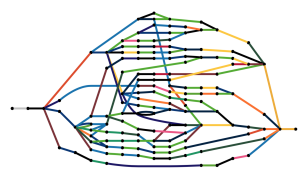
Efficient Negative Selection

Detection With the r-Pattern Graph

Detection with the transformed graph

1. Depth first search for paths matching the input string
2. If a match of length $r - 1$ is found, input string is nonself





Inefficient Negative Selection

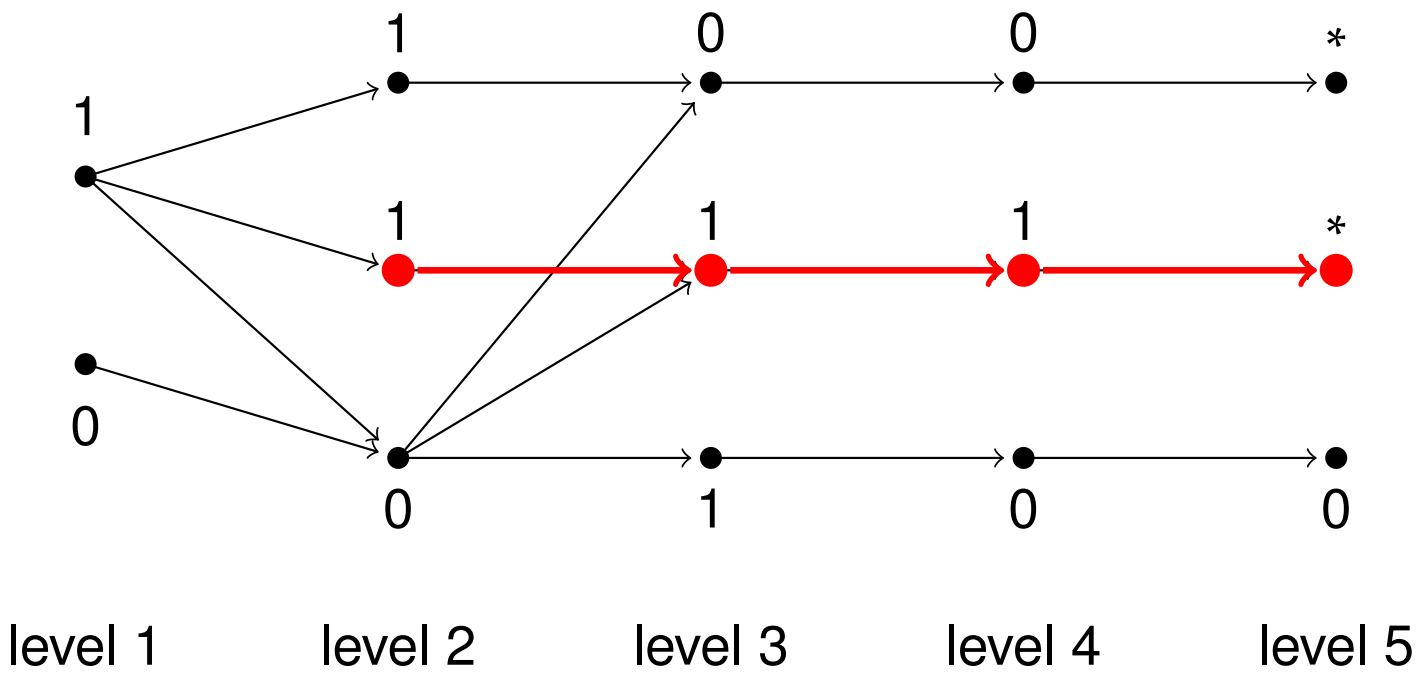
Efficient Negative Selection

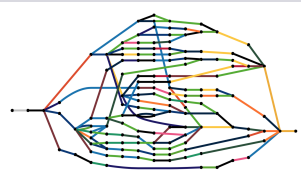
Detection With the r-Pattern Graph

Detection with the transformed graph

1. Depth first search for paths matching the input string
2. If a match of length $r - 1$ is found, input string is nonself

0 1 1 1 1 – nonself





Inefficient Negative Selection

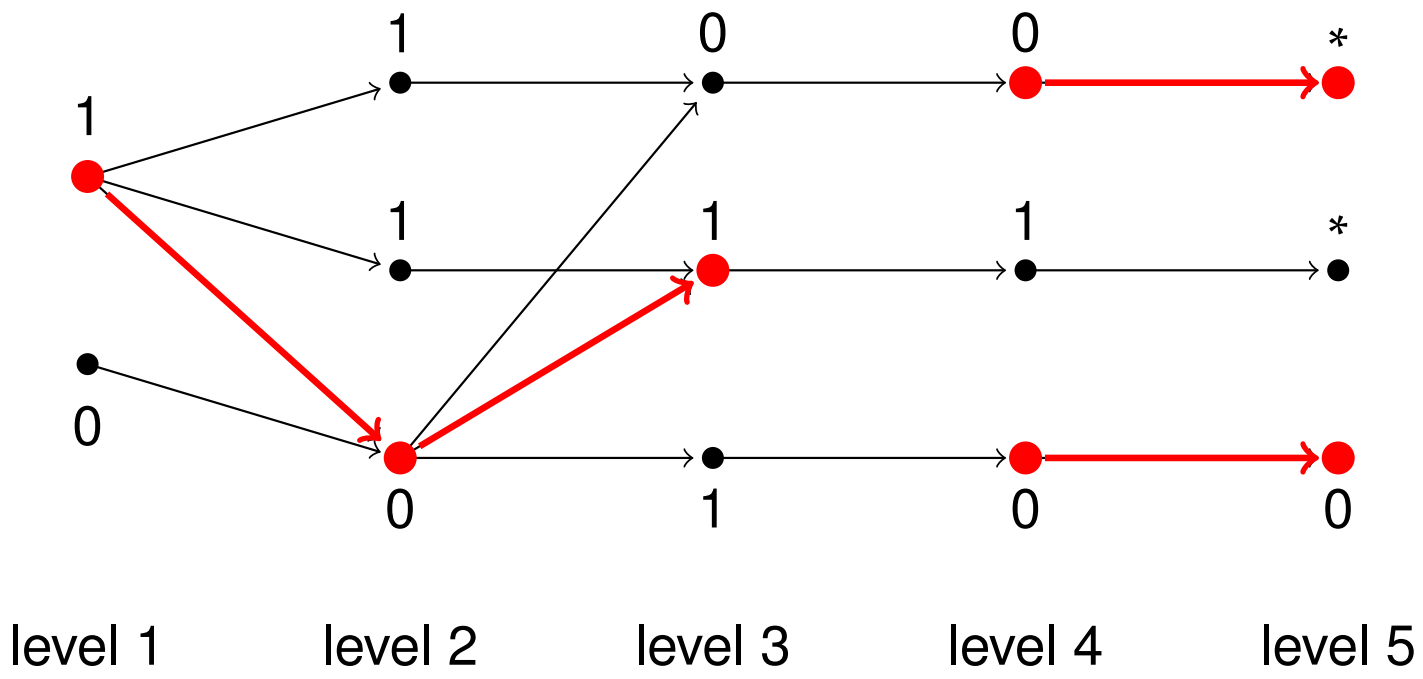
Efficient Negative Selection

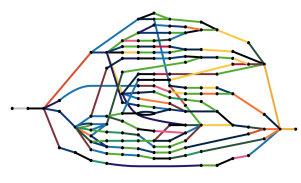
Detection With the r-Pattern Graph

Detection with the transformed graph

1. Depth first search for paths matching the input string
2. If a match of length $r - 1$ is found, input string is nonself

0 1 1 1 1 – nonself 1 0 1 0 0 – nonself





Inefficient Negative Selection

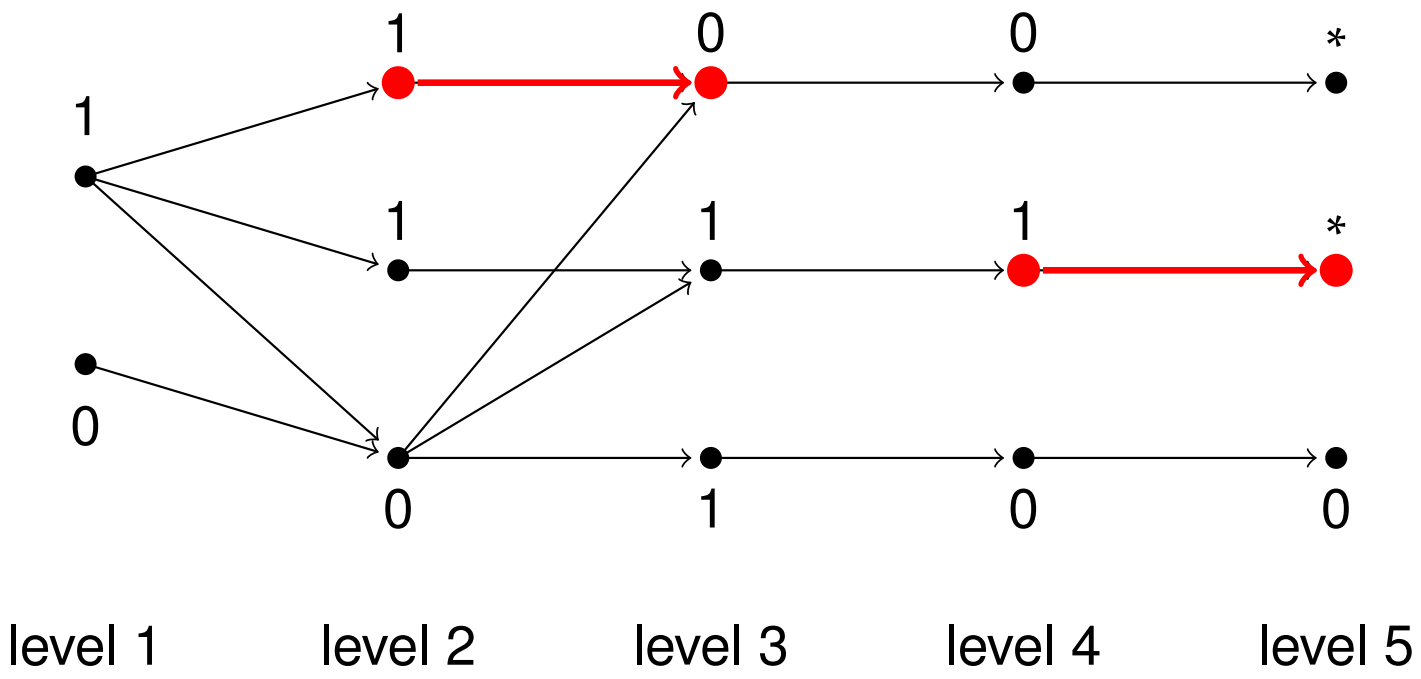
Efficient Negative Selection

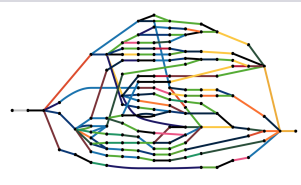
Detection With the r-Pattern Graph

Detection with the transformed graph

1. Depth first search for paths matching the input string
2. If a match of length $r - 1$ is found, input string is nonself

0 1 1 1 1 – nonself 1 0 1 0 0 – nonself 0 1 0 1 0 – self

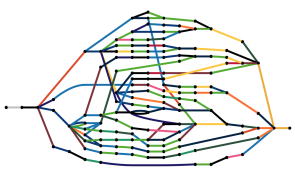




Inefficient Negative Selection

Efficient Negative Selection

Detection With the r-Pattern Graph

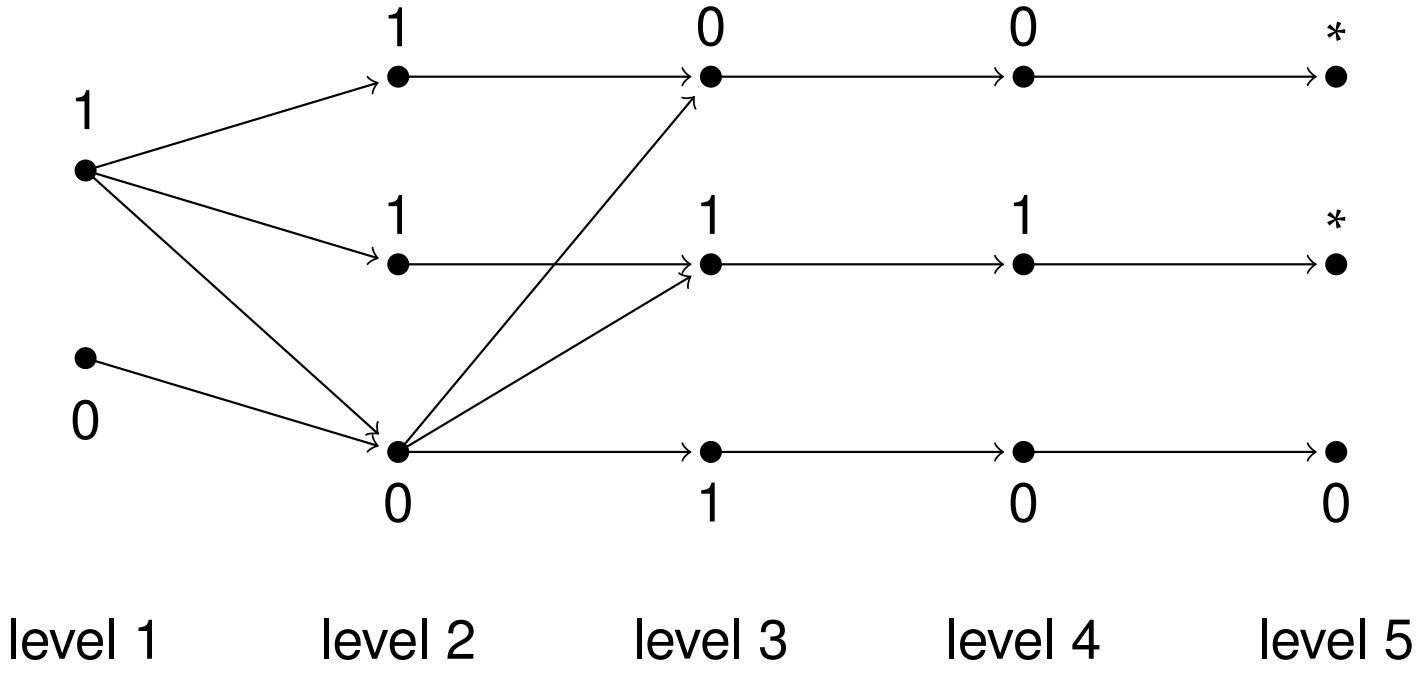


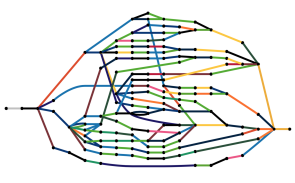
Inefficient Negative Selection
Efficient Negative Selection

Detection with the transformed graph

1. Depth first search for paths matching the input string
2. If a match of length $r - 1$ is found, input string is nonself

Set up: $O\left(|S|^3 L^3 r^3\right)$ Classify one string: $O\left(|V| + |E|\right) = O\left(|S|^2 L r^2\right)$





Inefficient Negative Selection

Efficient Negative Selection

The approach shown here can be, and has been, further generalized.

Substantial speedups can be achieved for many string matching functions, notably except the Hamming distance.

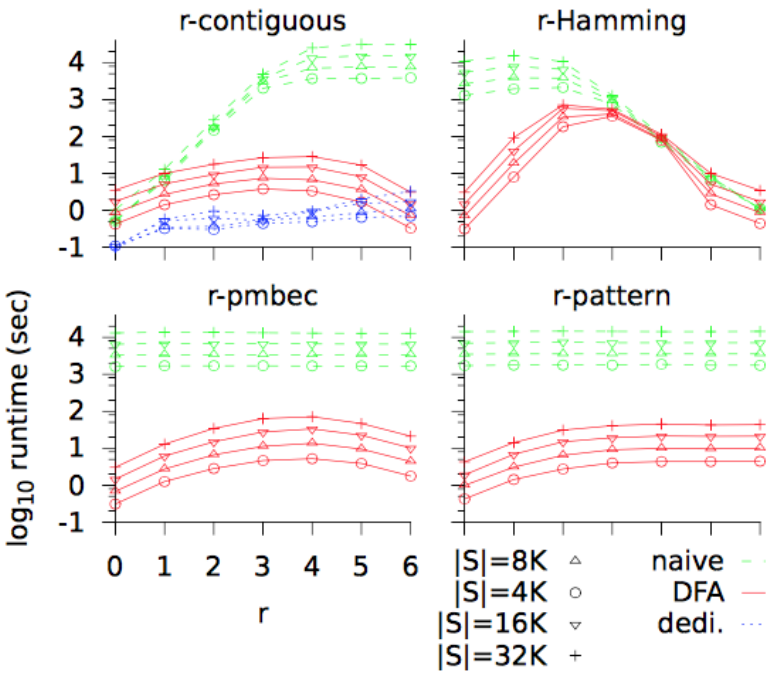
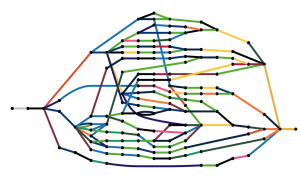


Figure 4: Empirical runtime analysis for repertoire counting after positive selection. We compare a naive implementation and the general DFA-based approach for $|S| \in \{4000, 8000, 16000, 32000\}$, $|\Sigma| = 20$ and $L = 6$. For r-contiguous detectors, we also run the dedicated *negative* selection implementation (dedi) by Textor [30] and convert the output to the positive selection count using the identity $|D[S^+]| = |\Sigma|^L - |D[S^-]|$.

(Textor, Dannenberg & Liśkiewicz, GECCO 2014)

Summary

- Literally implemented negative selection algorithms suck. Almost always.
- The key to efficient negative selection algorithms is using a compressed representation of the detector set.
- For some matching functions, this leads to an exponential speed-up.



Inefficient Negative Selection

Efficient Negative Selection