Typed functional genetic programming

Tomáš Křen

Supervisor: Roman Neruda

What is Genetic programming?

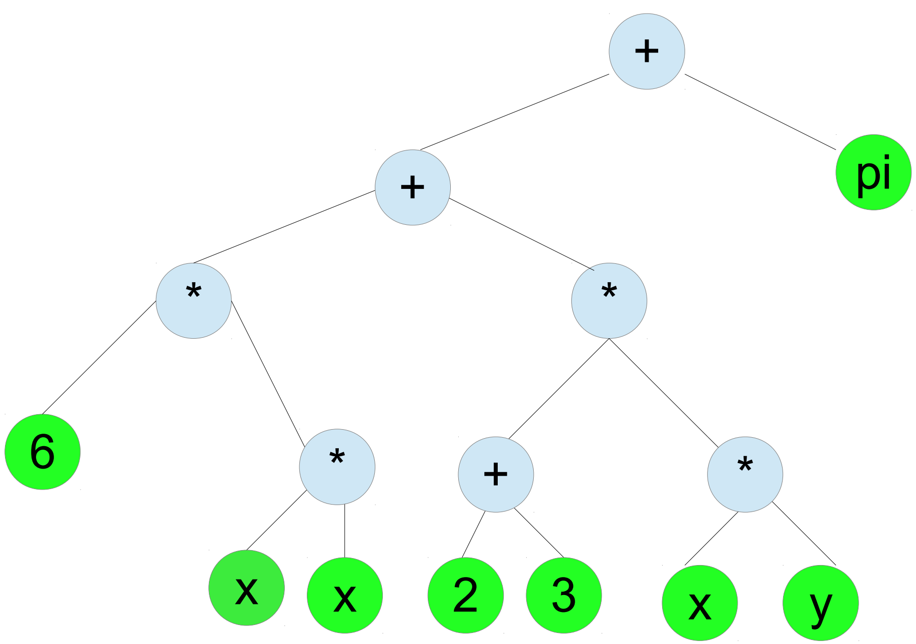
GP is a technique inspired by biological evolution that for a given problem tries to find computer programs able to solve that problem.

Author of GP: John **Koza** (1992)

- **Main inputs**:
  - Fitness function $(f : Program \rightarrow \mathbb{R}_0^+)$
  - Set of building symbols
- **Output:**
  - Programs (a simple S-expression)

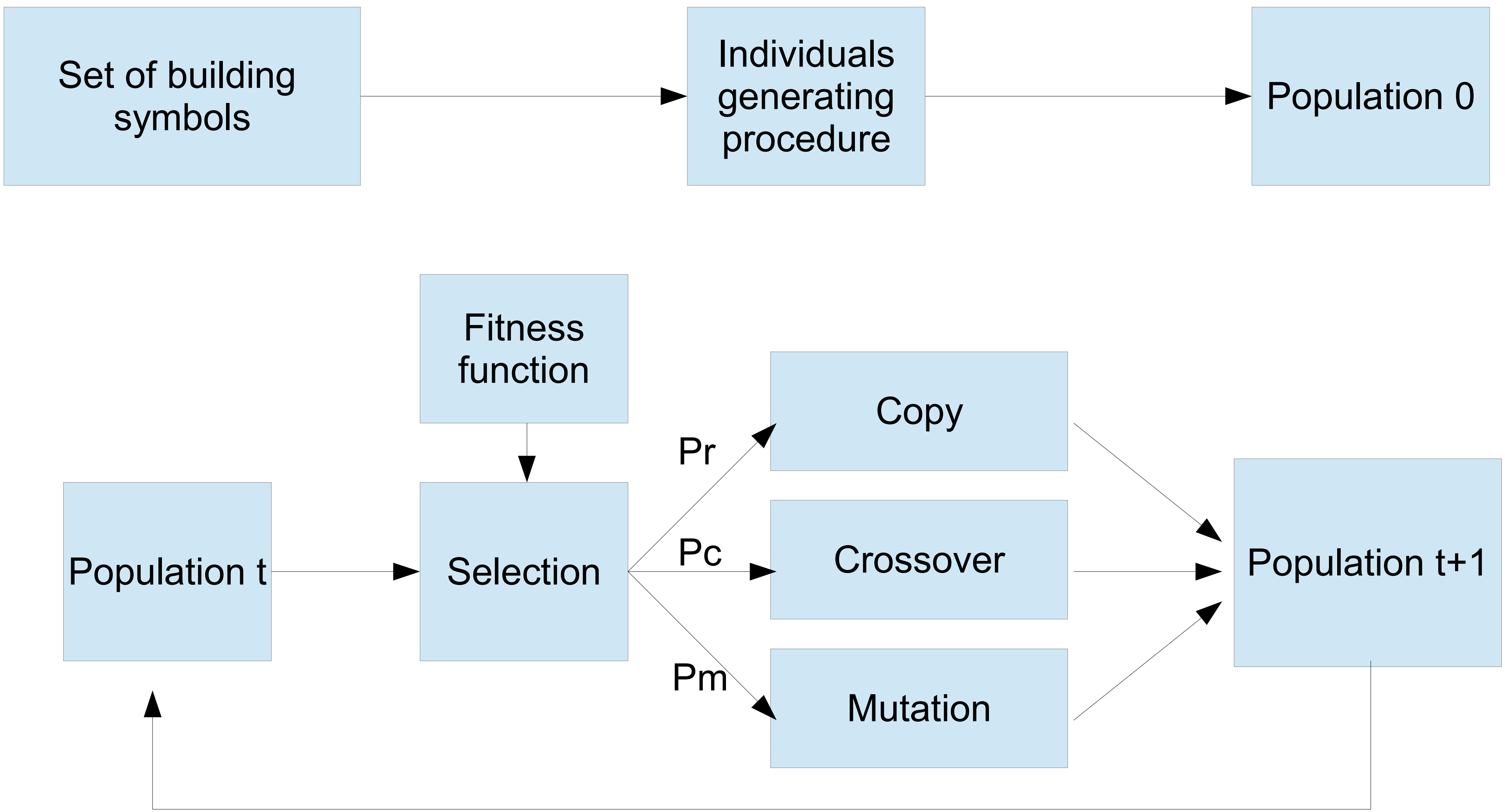
GP individual

- Syntactic tree of the program.
- Non-leaf nodes are function symbols. (set **F**)
- Leaf nodes are variables, constants or values. (set **T** … Terminals)
- Set of building symbols $\mathbf{\Gamma_0 = T \cup F}$




```
function(x,y){
  return 6*(x*x)+(2+3)*(x*y)+pi;
}
```

How it works?


Set of building symbols
Individuals generating procedure
Population 0
Fitness function
Population t
Selection
Pr
Pc
Pm
Copy
Crossover
Mutation
Population t+1

Types in GP

- Types help us overcome the *closure requirement*.
  - No longer need for “everything fitting into everything”.
- But they also establish new requirements
  - e.g. function arguments must obey type requirements...
  - These requirements make the programs more reasonable,
  - and reduce the search space.

Lambda calculus

- Simple yet powerful (mathematical) *functional programming* language
- It uses anonymous functions very often.
- Roughly speaking:

s-expressions + anonymous functions = lambda calculus

x may occure in this subtree...

λ *<var-name>* **.** *<body-expr>*

aka

function(*<var-name>***){ return** *<body-expr>***; }**

Benefits of using Functional programing for GP

- Complex and/or general programming constructs can be described as higher-order functions

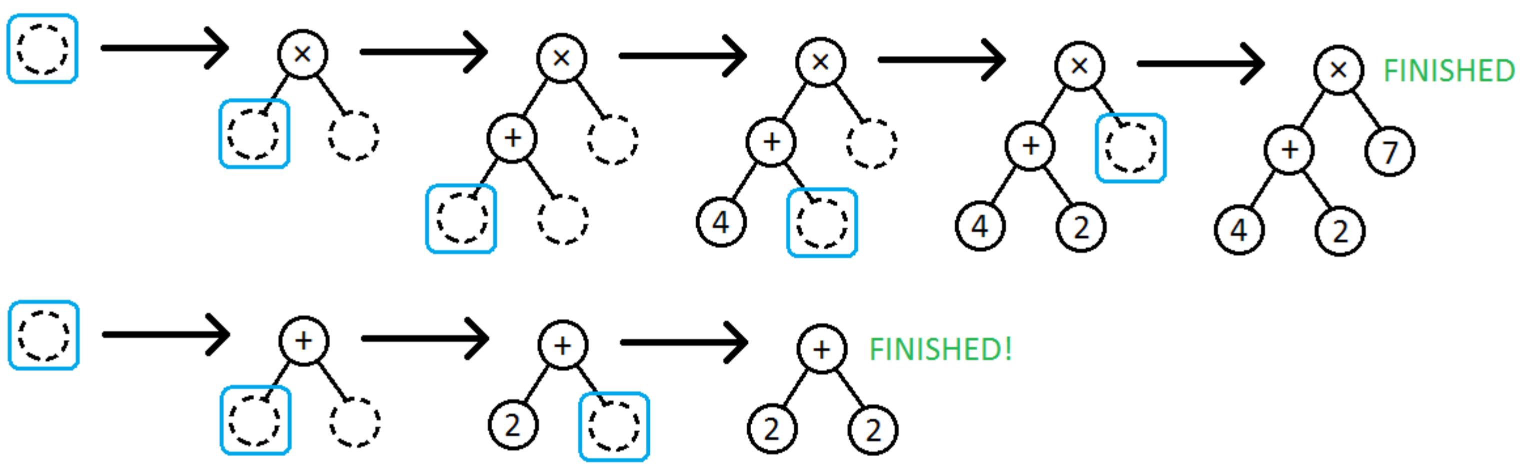
- Types provide rigorous way to talk about (sub)programs and to enforce constraints.

Curry-Howard correspondence

	Inductive	*Compromise*	*Deductive*
	Standard GP	**Typed functional GP**	**Automatic Theorem Prover**
Generating	Trivial generating.	"Hybrid generating" combing both aproaches.	Hard. Generating is the only task of a prover.
What to do with generated program?	**Selection & Genetic operators** Genarated programs are stupid. The creativity of GP lies in the iterative evolution toward better performance.	Generalized operators for typed functional GP	Nothing, we are ok with any prove.
	No type system	Moderately powerful type system	Very powerful type system

Standard generating procedure

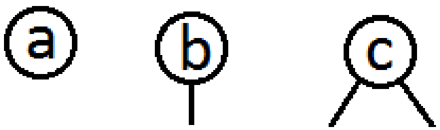
- works in separate iterations
  - In each iteration one tree individual is generated.



- We can do this differently:
  - “Generating of shared parts can be shared.”

Exhaustive enumeration of individuals

INPUT

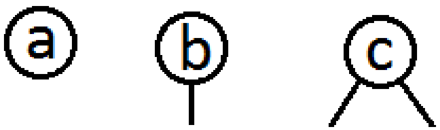


e.g. T = {a}, F = {b:1 arg, c:2 args}

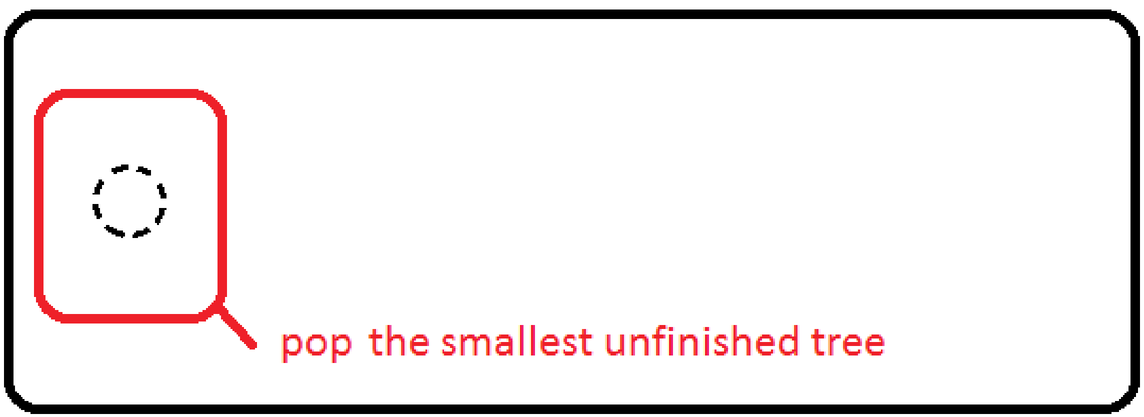
PRIORITY QUEUE

OUTPUT

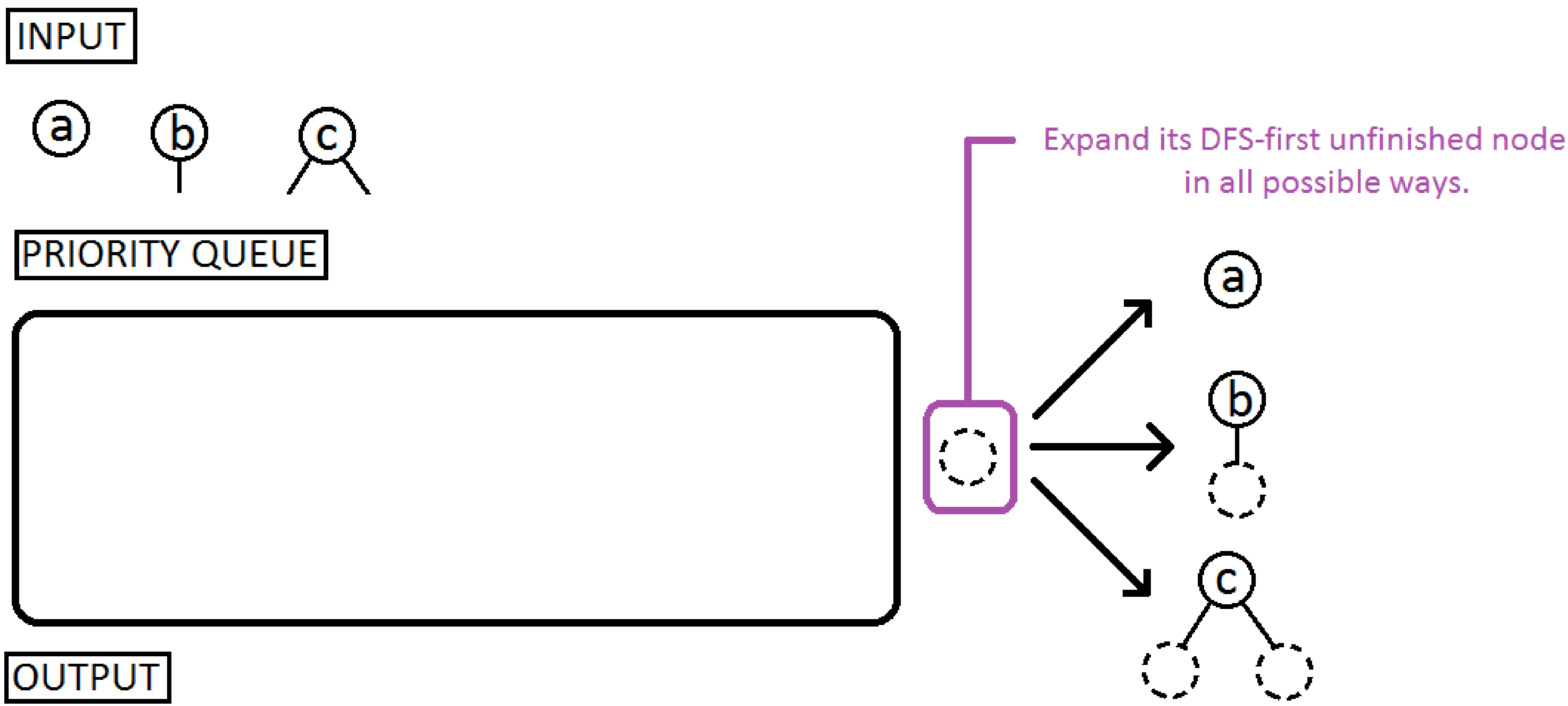
INPUT

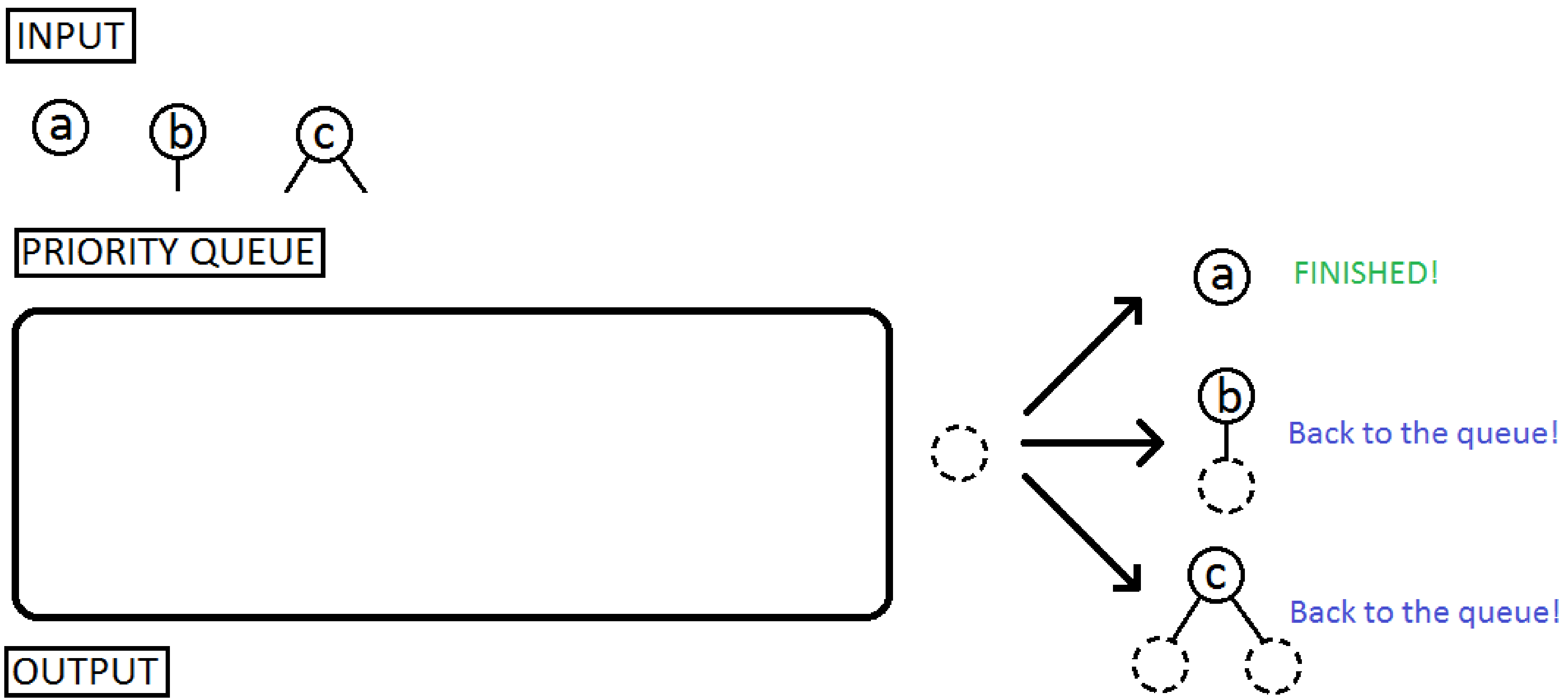


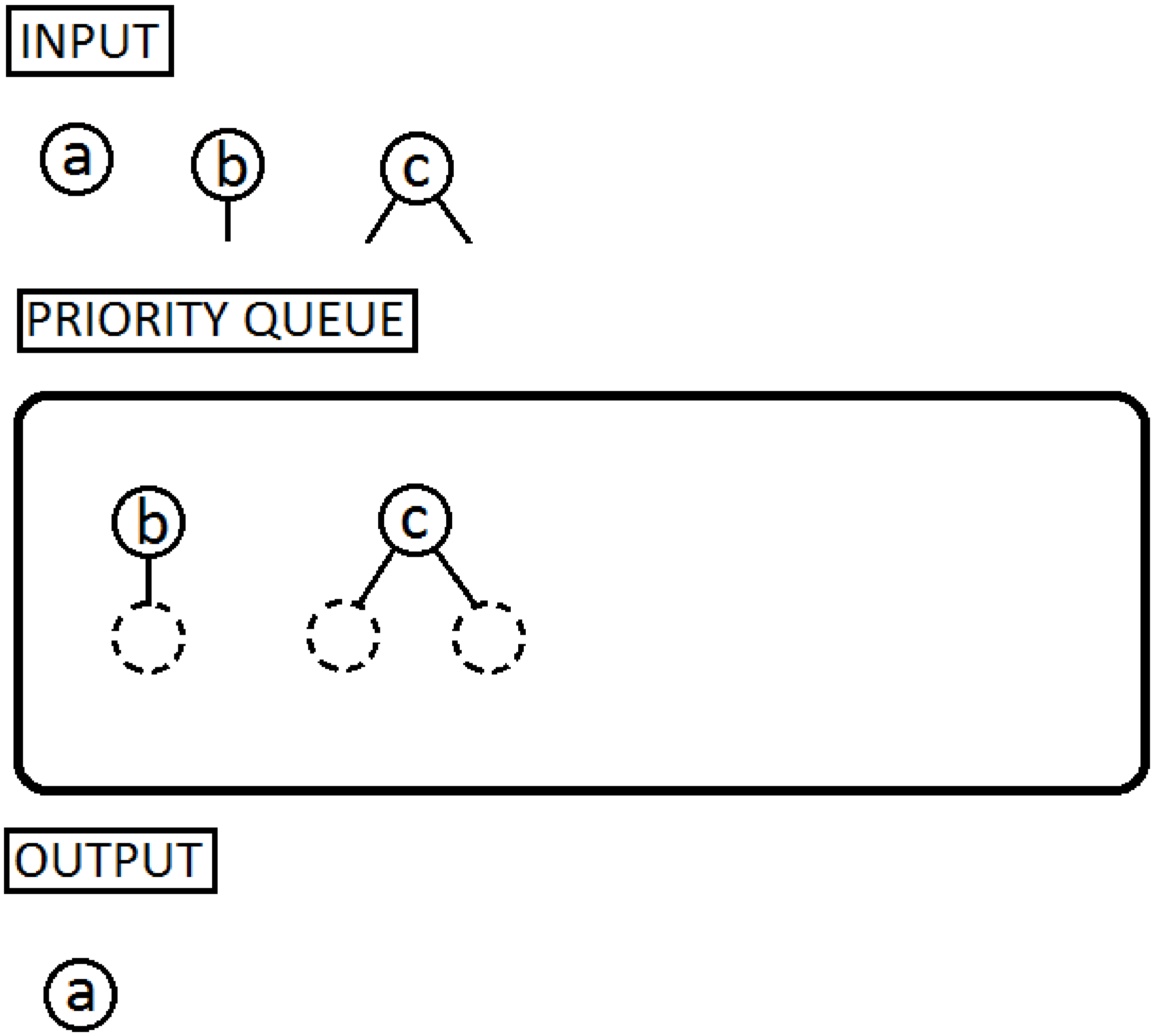
PRIORITY QUEUE

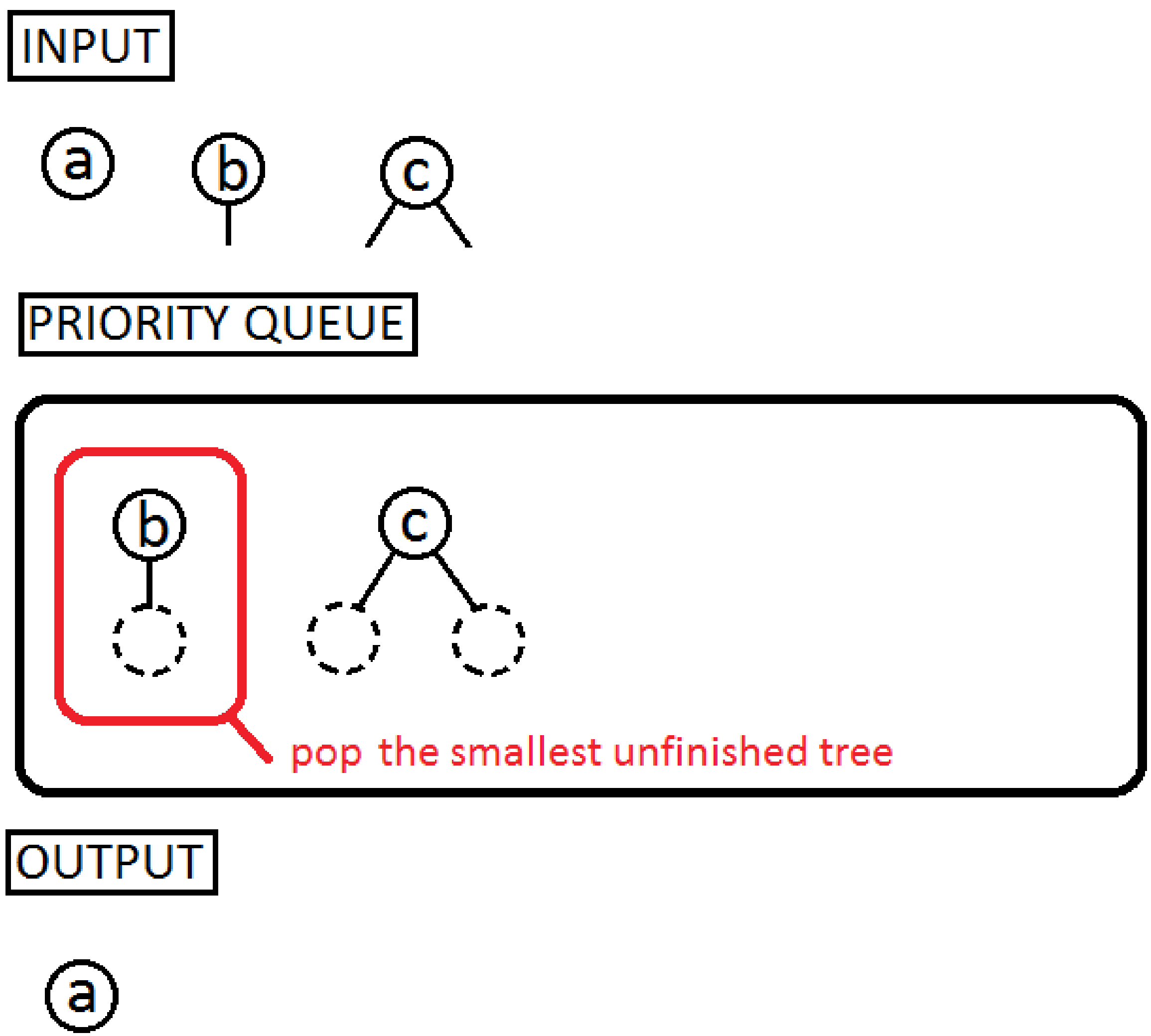


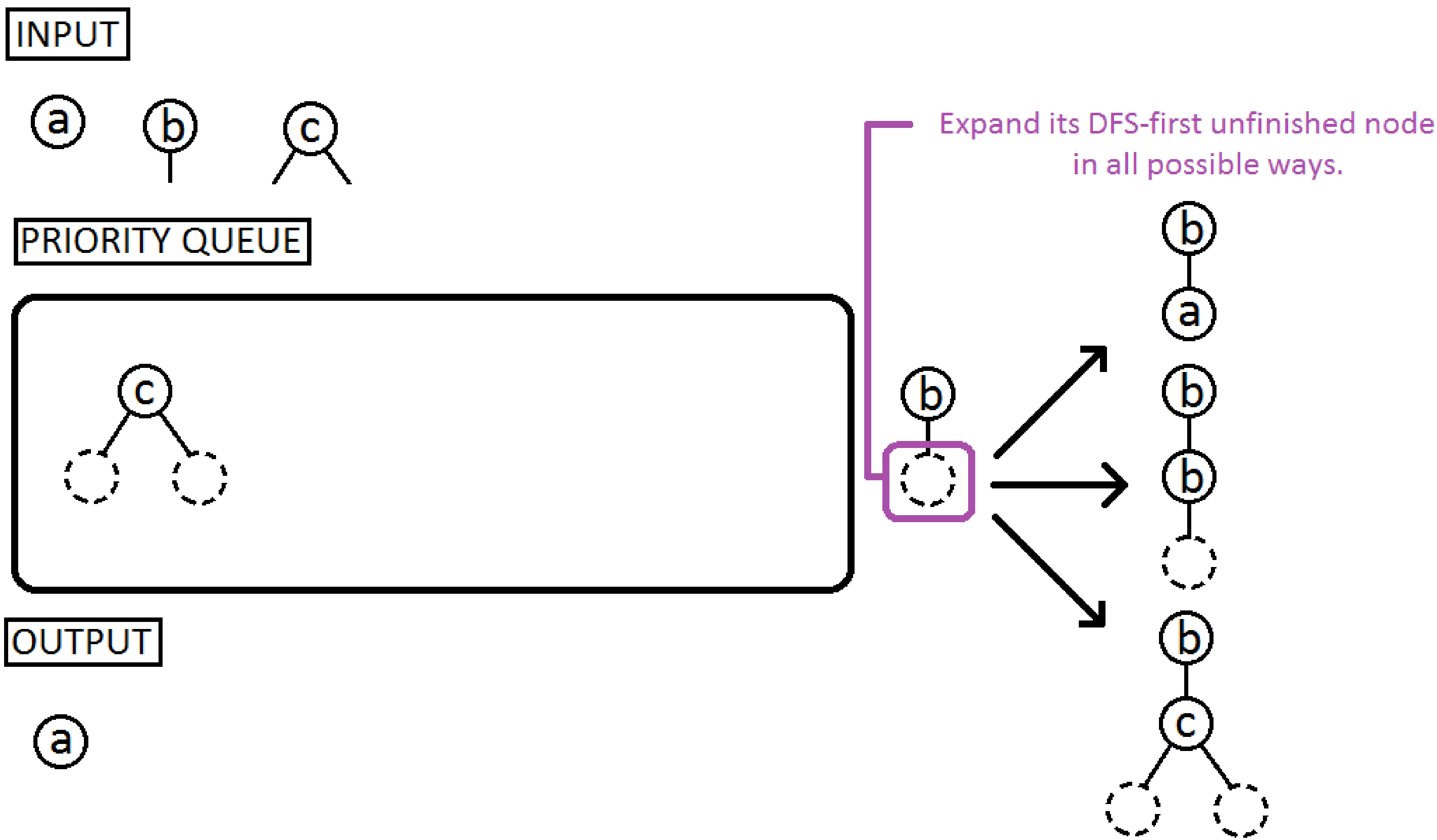
OUTPUT

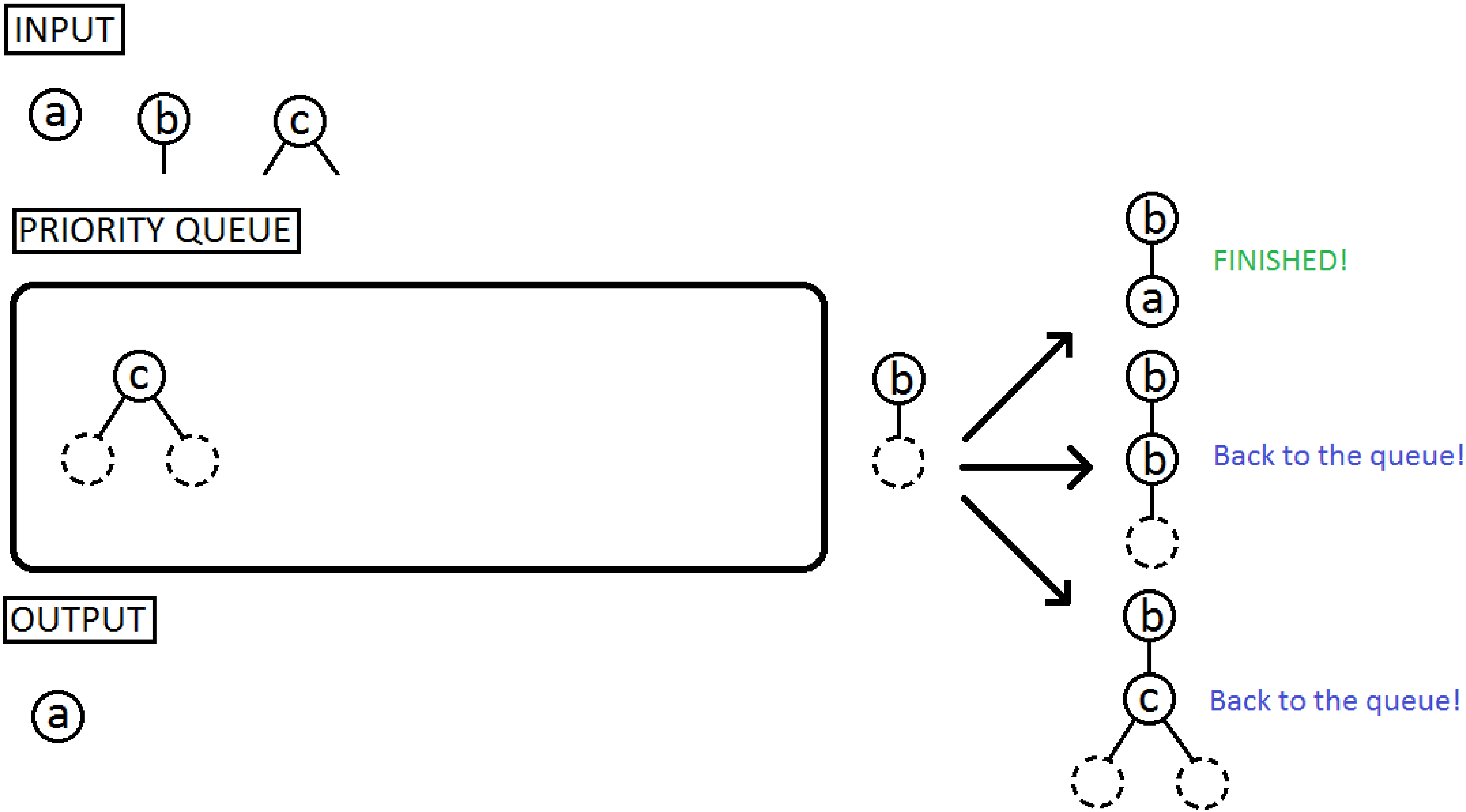

INPUT
a
b
c
PRIORITY QUEUE
OUTPUT
Expand its DFS-first unfinished node
in all possible ways.
a
b
c

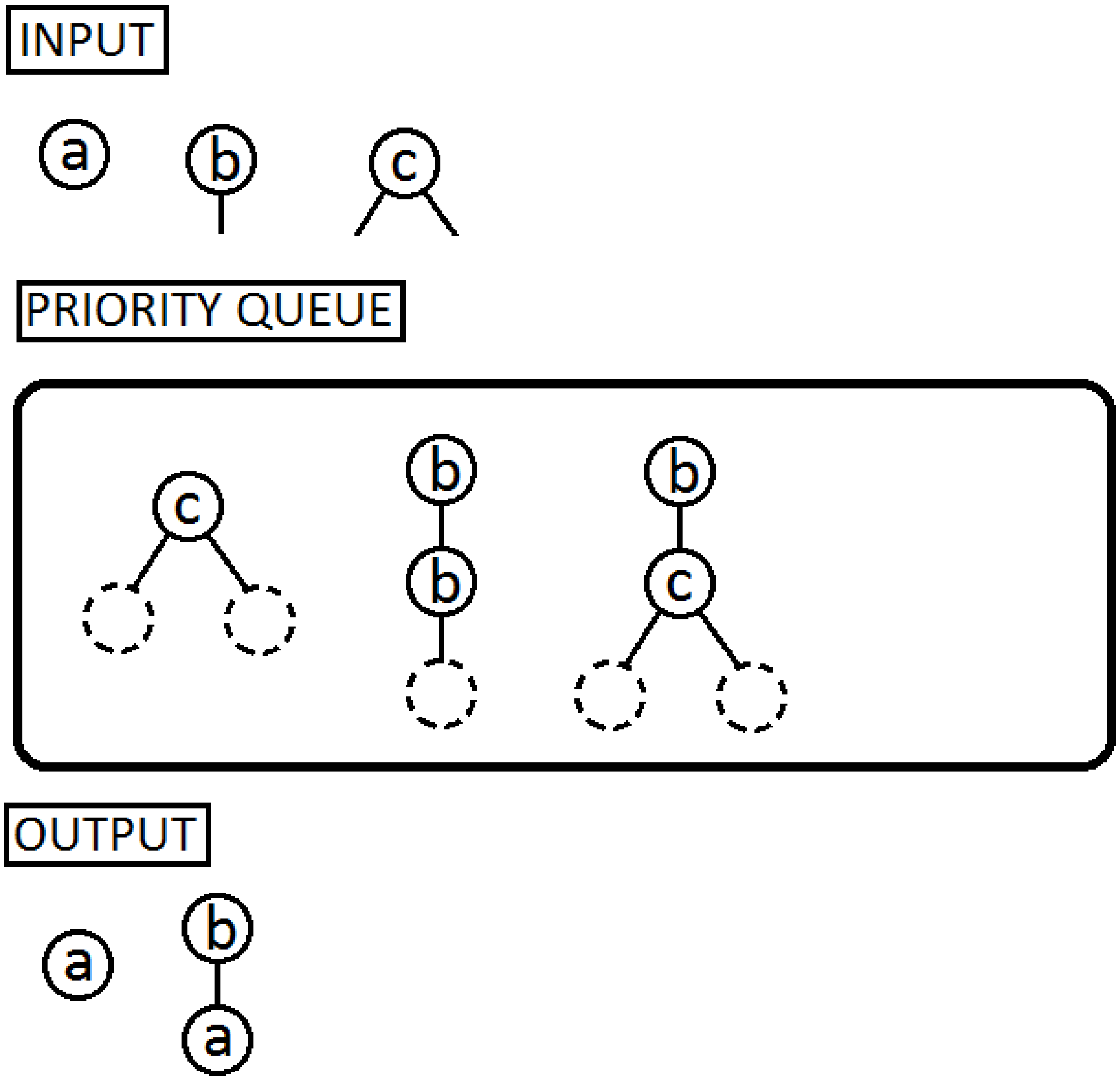

INPUT
a
b
c
PRIORITY QUEUE
OUTPUT
a
FINISHED!
b
Back to the queue!
c
Back to the queue!


INPUT
a
b
c
PRIORITY QUEUE
b
c
OUTPUT
a


INPUT
a
b
c
PRIORITY QUEUE
b
c
pop the smallest unfinished tree
OUTPUT
a


INPUT
a
b
c
PRIORITY QUEUE
c
OUTPUT
a
Expand its DFS-first unfinished node
in all possible ways.
b
b
a
b
b
b
c

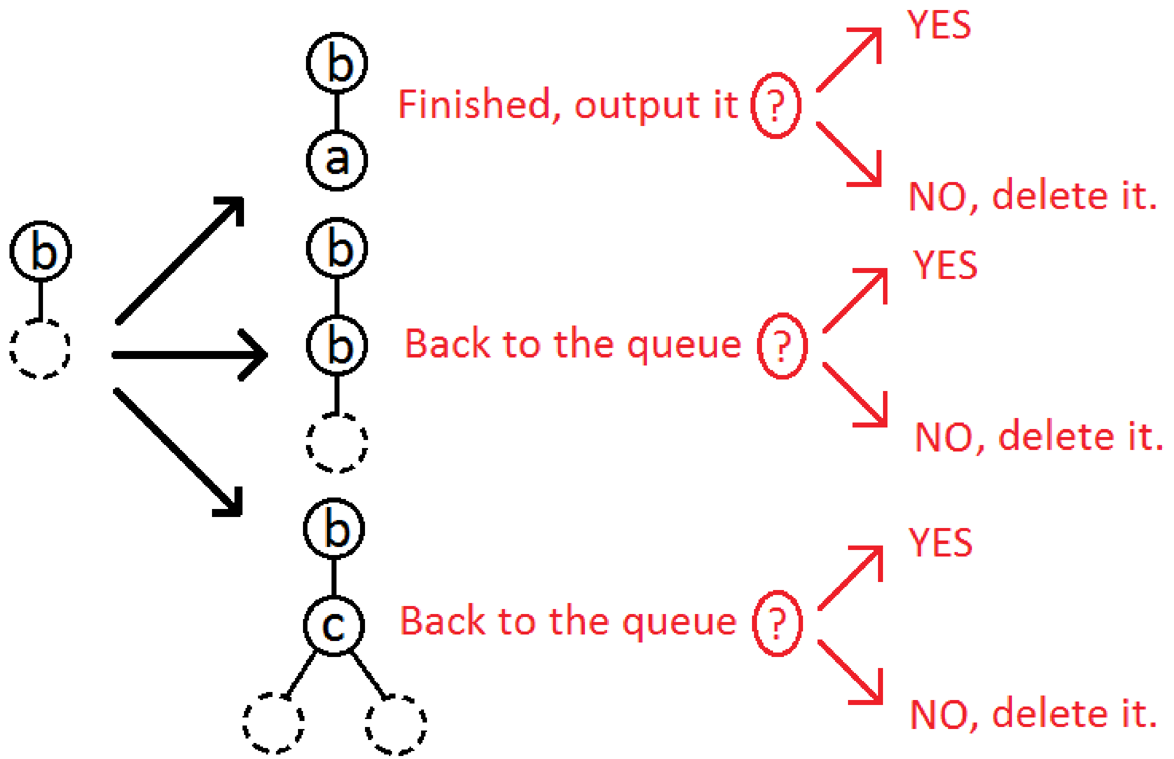

INPUT
a
b
c
PRIORITY QUEUE
c
b
b
a
FINISHED!
b
b
Back to the queue!
b
c
Back to the queue!
OUTPUT
a



INPUT
a
b
c
PRIORITY QUEUE
c
b
b
b
c
OUTPUT
a
b
a

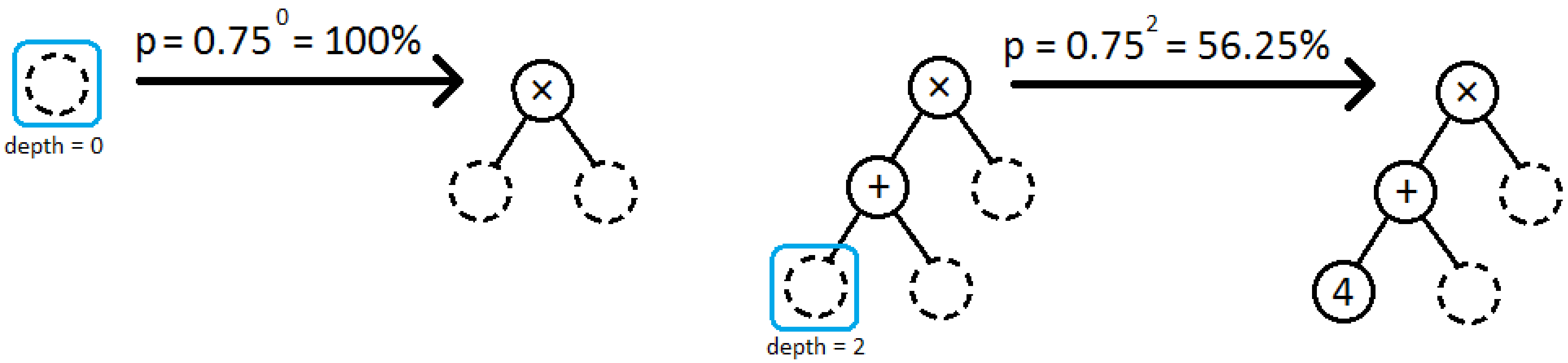
How to make enumeration more random?

- We add a new step deciding what to do with an expanded tree:
  - keep it,
  - or delete it?
- We call this additional decision procedure a *generating strategy.*
  - *"Keep all"* strategy = exhaustive enumeration
  - *"Delete all but one"* strategy = standard generating approach

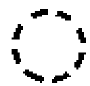
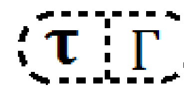
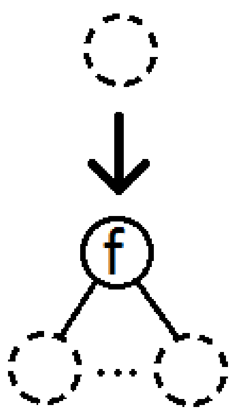
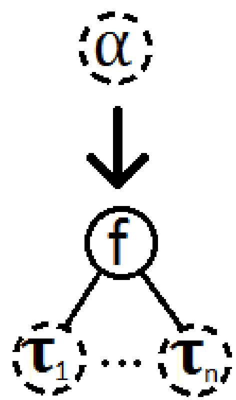
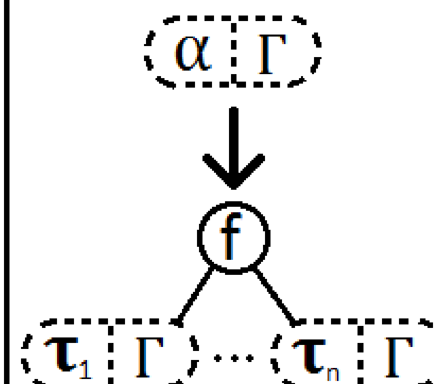
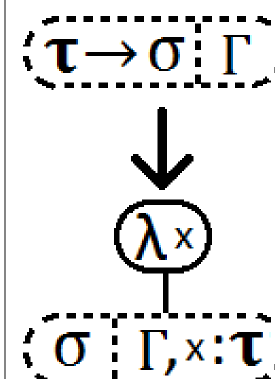


Our *geometric* strategy

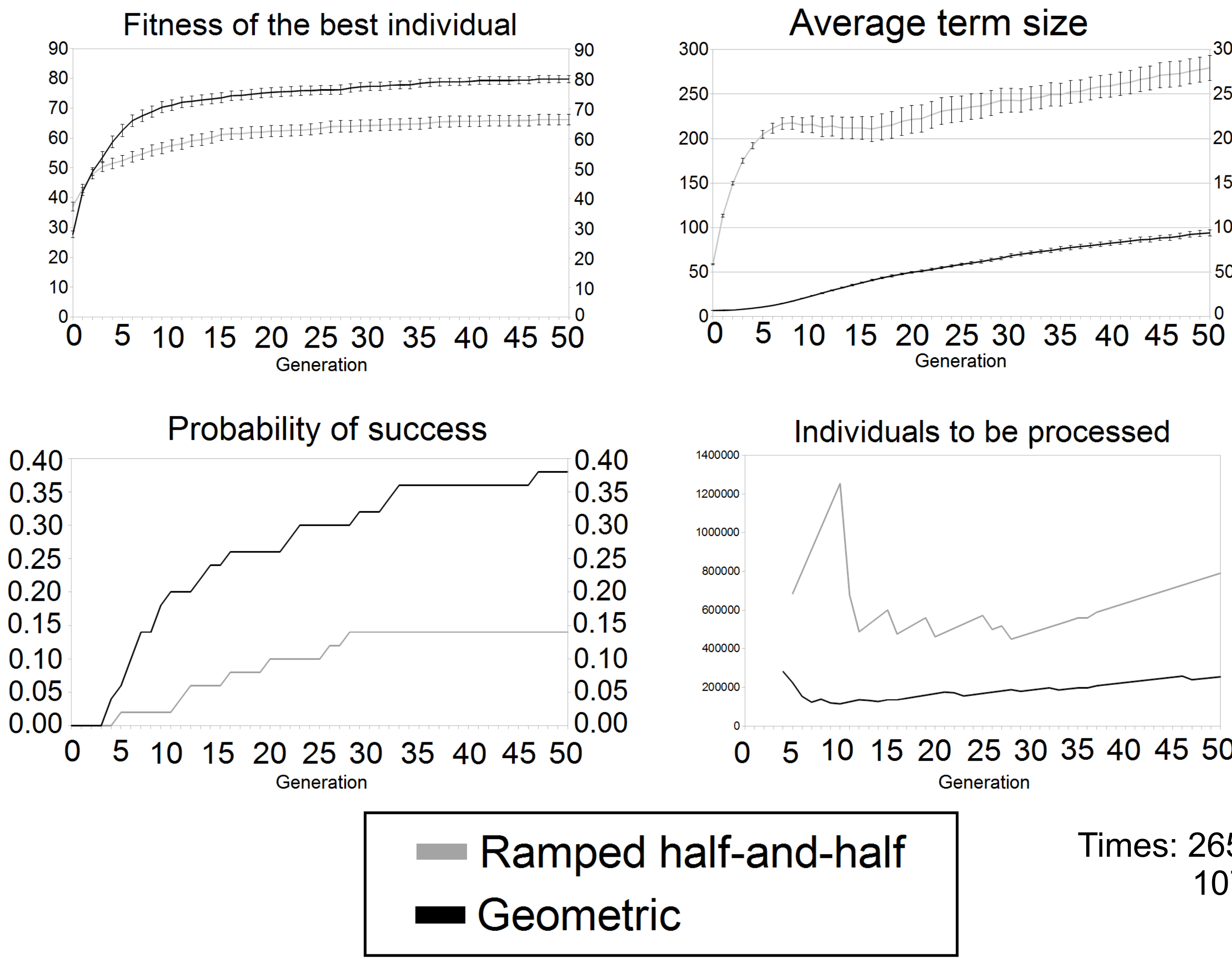
- It puts an expanded tree back to the queue with probability $\mathbf{p = q^{depth}}$
- Where **q** is a constant, we used **q = 0.75**
- And **depth** is depth of the expanded node



Generalization for simply typed lambda calculus

	No types	Types, no contexts	Simply typed lambda calculus
Unfinished node			
Expansion(s)		 $f : \underbrace{\tau_1 \to \cdots \to \tau_n}_{\text{inputs types}} \to \underbrace{\alpha}_{\text{ouput type}}$	*atomic types:*  $(f:\tau_1\to\cdots\to\tau_n\to\alpha)\in\Gamma$ *function types:* 

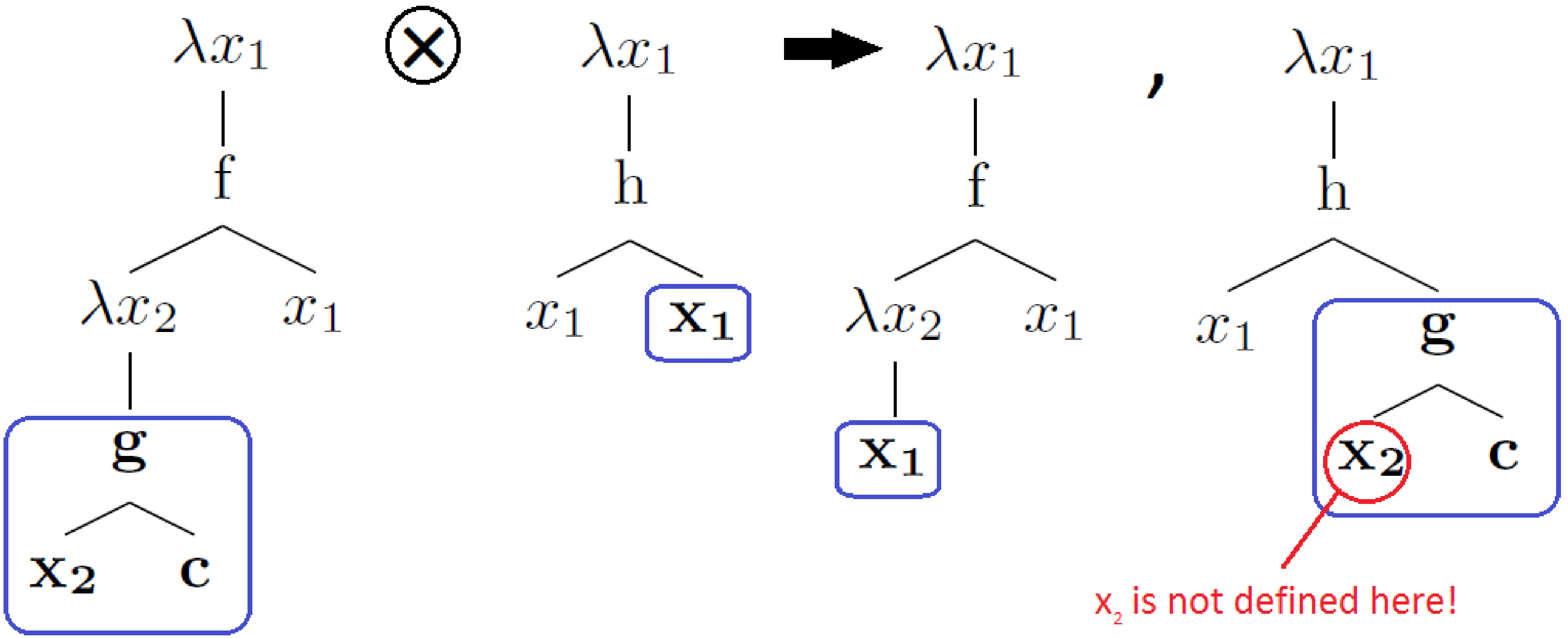
Artificial ant problem



Times: 265 minutes
107 minutes

Crossover operator for lambda terms

- Generalization of simple tree swapping crossover
- We need to swap subtrees with a same type
  - ..but that is simple
- Local variables cause the trouble!



Abstraction elimination

- An algorithm for getting rid of local variables and anonymous function
  - **Input:** an arbitrary lambda term
  - **Output:** equivalent S-expression *(with no local variables or anonymous functions)* that may contain additional new function nodes **S**,**K** and **I** which are defined as:

$$\mathbf{S} = \lambda\, f\, g\, x\, .\, f\, x\, (g\, x)$$
$$\mathbf{K} = \lambda\, x\, y\, .\, x$$
$$\mathbf{I} = \lambda\, x\, .\, x$$

i.e.

"**function**(f,g,x){ **return** f(x, g(x)) }"

"**function**(x,y){ **return** x }"

"**function**(x){ **return** x }"

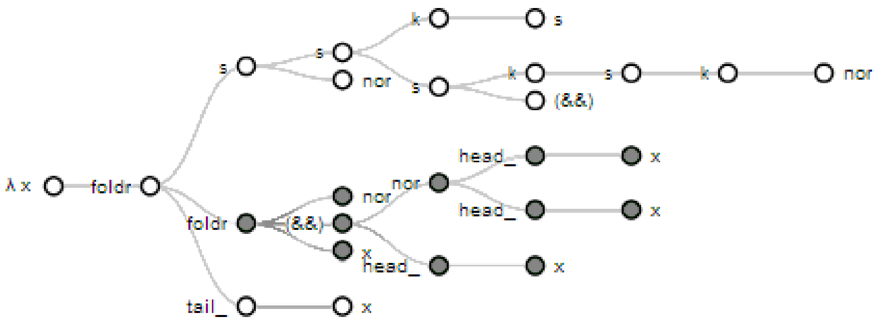
Hybrid crossover

- Each generated term is transformed by abstraction elimination
- So now all terms are typed S-expressions
- So now we only need to swap subtrees with the same type

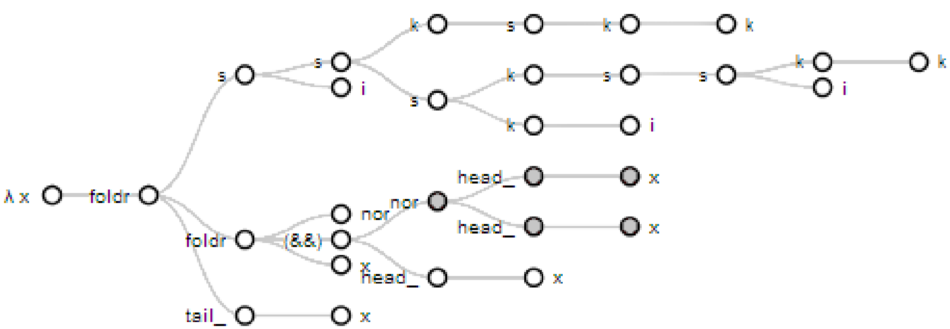
Unpacking crossover

- All terms are kept in small $\beta\eta$-normal form
- … and transformed right before crossover
- After the tree swapping both children are again normalized
- So the quadratic increase is only temporary

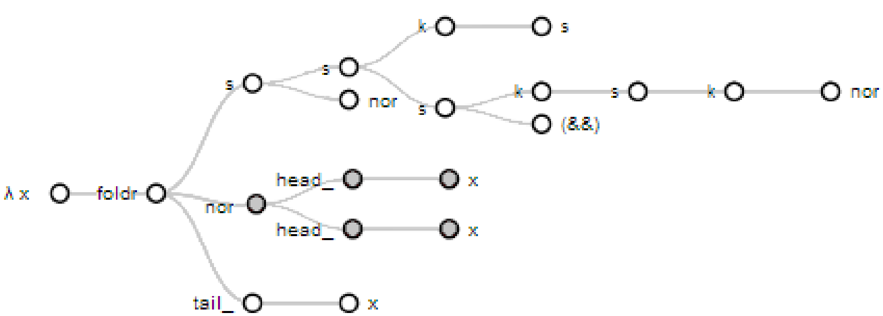
Even parity problem



Parent #1, fitness: 0.8125



Parent #2, fitness: 0.5



Child #1, fitness: 1

$$\lambda x\ .\ foldr\ (\mathbf{S}(\mathbf{S}(\mathbf{K}\ \mathbf{S})(\mathbf{S}(\mathbf{K}(\mathbf{S}(\mathbf{K}\ nor)))and))nor)\ (nor\ (head'\ x)\ (head'\ x))\ \ (tail'\ x)$$

$$=_{\beta\eta}$$

$$\lambda x\ .\ foldr\ (\lambda y\ z\ .\ nor\ (and\ y\ z)\ (nor\ y\ z))\ (nor\ (head'\ x)\ (head'\ x))\ \ (tail'\ x)$$

Which is equivalent to:

$$\lambda x\ .\ foldr\ xor\ (not\ (head'\ x))\ \ (tail'\ x)$$

GP approach	$I(M,i,z)$
PolyGP	14,000
Our approach (hybrid)	**28,000**
GP with Combinators	58,616
GP with Iteration	60,000
Our approach (unpacking)	**114,000**
Generic GP	220,000
OOGP	680,000
GP with ADFs	1,440,000

Results comparison

Articles

- Generating Lambda Term Individuals in Typed Genetic Programming Using Forgetful A*
  - IEEE WCCI 2014, Beijing
- Utilization of Reductions and Abstraction Elimination in Typed Genetic Programming
  - GECCO 2014, Vancouver

Future work

- Implement more general type system
  - Hindley–Milner type system
  - Hindley–Milner enriched with Type classes
    - This enriches the logic with predicates.

- Design an interesting problem for this system
  - Problems around simulation of simple economy from the multi-agent point of view