Learning efficient logic programs

Input	Output
[s,h,e,e,p]	e
[a,l,p,a,c,a]	a
[c,h,i,c,k,e,n]	?

Input	Output
[s,h,e,e,p]	e
[a,l,p,a,c,a]	a
[c,h,i,c,k,e,n]	**c**

```
%% E+
f([s,h,e,e,p],e).
f([a,l,p,a,c,a],a).
f([c,h,i,c,k,e,n],c).

%% E-
f([s,h,e,e,p],p).
f([a,l,p,a,c,a],l).

%% BK
head([H|_],H).
tail([_|T],T).
length(A,B):- ..
element(A,B):- ..
mergesort(A,B):- ..
```

```
%% metagol
f(A,B):-head(A,B),tail(A,C),element(C,B).
f(A,B):-tail(A,C),f(C,B).
```

```prolog
%% metagol
f(A,B):-head(A,B),tail(A,C),element(C,B).
f(A,B):-tail(A,C),f(C,B).
```

```python
def f(xs):
    h = head(xs)
    t = tail(xs)
    if h is in t:
        return h
    return f(t)
```

```
%% alternative
f(A,B):-mergesort(A,C),f1(C,B).
f1(A,B):-head(A,B),tail(A,C),head(C,B).
f1(A,B):-tail(A,C),f1(C,B).
```

```prolog
%% alternative
f(A,B):-mergesort(A,C),f1(C,B).
f1(A,B):-head(A,B),tail(A,C),head(C,B).
f1(A,B):-tail(A,C),f1(C,B).
```

```python
def f(xs):
    ys = sorted(xs)
    return f1(ys)

def f1(xs):
    h = head(xs)
    t = tail(xs)
    hh = head(t)
    if h == hh:
        return h
    return f1(t)
```

Idea

Input

- examples **E**
- background knowledge **B**
- **cost** : Program × Example → N

Idea

1. Learn any program **H**
2. Repeat while possible:
    a. Learn program **H’** where **max_cost(H’,E)** < **max_cost(H,E)**
    b. **H=H’**
3. Return **H**

Metagol

```
prove([],P,P).

prove([Atom|Atoms],P1,P2):-
    prove_aux(Atom,P1,P3),
    prove(Atoms,P3,P2).

prove_aux(Atom,P,P):-
    call(Atom).

prove_aux(Atom,P1,P2):-
    metarule(Atom,Body,Subs),
    save(Subs,P1,P3),
    prove(Body,P3,P2).
```

Metagol

```
prove([],P,P).

prove([Atom|Atoms],P1,P2):-
    prove_aux(Atom,P1,P3),
    prove(Atoms,P3,P2).

prove_aux(Atom,P,P):-
    call(Atom).

prove_aux(Atom,P1,P2):-
    metarule(Atom,Body,Subs),
    save(Subs,P1,P3),
    prove(Body,P3,P2).
```

Metagol

```
prove([],P,P).

prove([Atom|Atoms],P1,P2):-
    prove_aux(Atom,P1,P3),
    prove(Atoms,P3,P2).

prove_aux(Atom,P,P):-
    call(Atom).

prove_aux(Atom,P1,P2):-
    metarule(Atom,Body,Subs),
    save(Subs,P1,P3),
    prove(Body,P3,P2).
```

Metagol

```
prove([],P,P).

prove([Atom|Atoms],P1,P2):-
    prove_aux(Atom,P1,P3),
    prove(Atoms,P3,P2).

prove_aux(Atom,P,P):-
    call(Atom).

prove_aux(Atom,P1,P2):-
    metarule(Atom,Body,Subs),
    save(Subs,P1,P3),
    prove(Body,P3,P2).
```

Metaopt

```
prove([],P,P,C,C).

prove([Atom|Atoms],P1,P2,C1,C2):-
    prove_aux(Atom,P1,P3,C1,C3),
    prove(Atoms,P3,P2,C3,C2).

prove_aux(Atom,P,P,C1,C2):-
    pos_cost(Atom,Cost).
    C2 is C1+Cost,
    bound(MaxCost),
    C2 < MaxCost.

prove_aux(Atom,P1,P2,C1,C2):-
    metarule(Atom,Body,Subs),
    save(Subs,P1,P3),
    C3 is C1+1,
    prove(Body,P3,P2,C3,C2).
```

Metaopt

```
prove([],P,P,C,C).

prove([Atom|Atoms],P1,P2,C1,C2):-
    prove_aux(Atom,P1,P3,C1,C3),
    prove(Atoms,P3,P2,C3,C2).

prove_aux(Atom,P,P,C1,C2):-
    pos_cost(Atom,Cost).
    C2 is C1+Cost,
    bound(MaxCost),
    C2 < MaxCost.

prove_aux(Atom,P1,P2,C1,C2):-
    metarule(Atom,Body,Subs),
    save(Subs,P1,P3),
    C3 is C1+1,
    prove(Body,P3,P2,C3,C2).
```

Metaopt

```
prove([],P,P,C,C).

prove([Atom|Atoms],P1,P2,C1,C2):-
    prove_aux(Atom,P1,P3,C1,C3),
    prove(Atoms,P3,P2,C3,C2).

prove_aux(Atom,P,P,C1,C2):-
    pos_cost(Atom,Cost).
    C2 is C1+Cost,
    bound(MaxCost),
    C2 < MaxCost.

prove_aux(Atom,P1,P2,C1,C2):-
    metarule(Atom,Body,Subs),
    save(Subs,P1,P3),
    C3 is C1+1,
    prove(Body,P3,P2,C3,C2).
```

Iterative descent

1. Learn any program **H** with minimal clauses
2. Repeat while possible:
    a. Learn program **H’** where **max_cost(H’,E)** < **max_cost(H,E)**
    b. **H=H’**
3. Return **H**

Metaopt prunes as it learns

Tree cost

Positive examples: size of the leftmost successful branch

Tree cost

Positive examples: size of the leftmost successful branch

```
pos_cost(Atom,Cost):-
    statistics(inferences,I1),
    call(Atom),
    statistics(inferences,I2),
    Cost is I2-I1.
```

Tree cost

Negative examples: size of the finitely-failed SLD-tree

Tree cost

Negative examples: size of the finitely-failed SLD-tree

```
neg_cost(Atom,Cost):-
    statistics(inferences,I1),
    \+ call(Atom),
    statistics(inferences,I2),
    Cost is I2-I1.
```

Tree cost

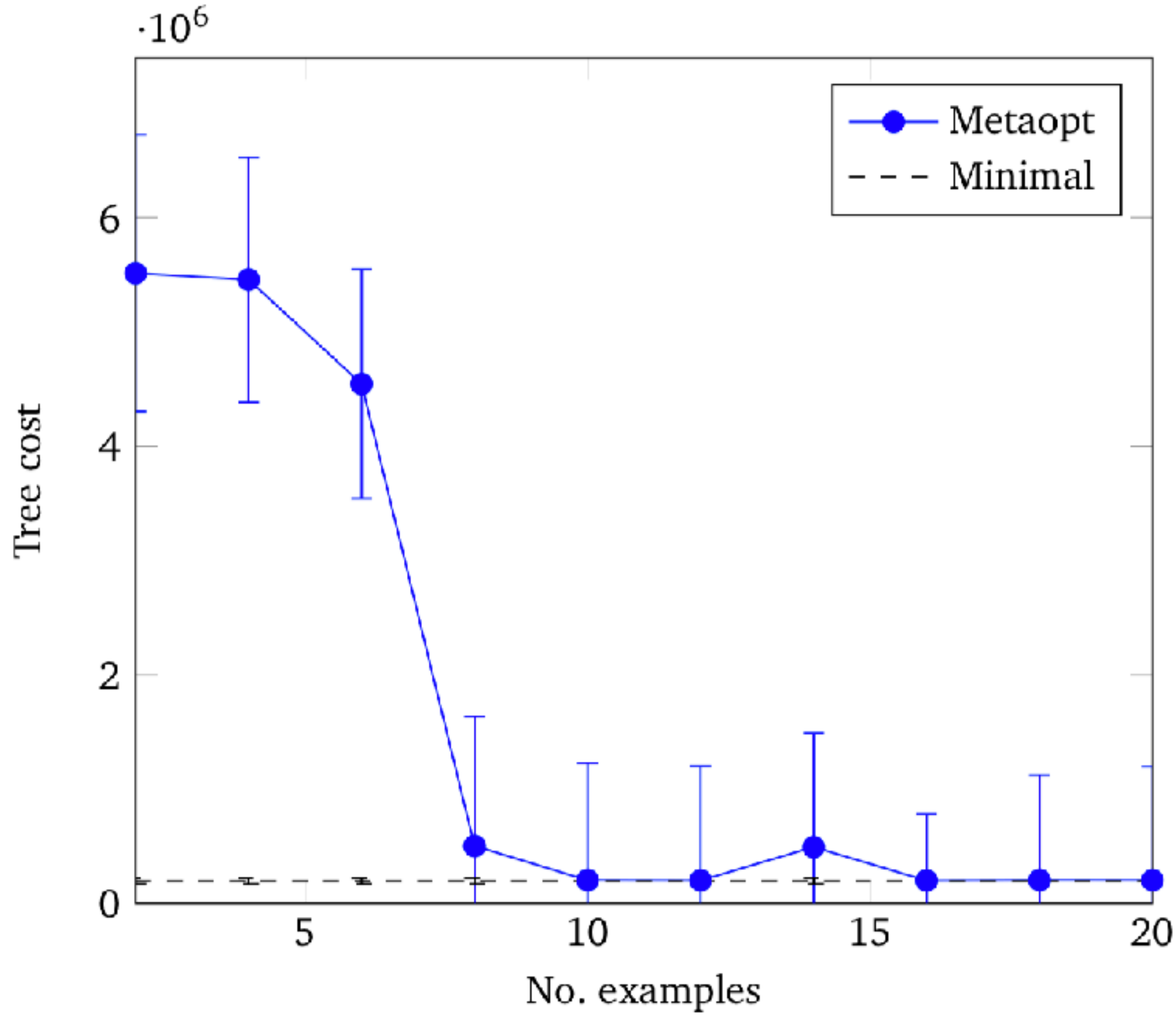
- any arity logics
- no user-supplied costs
- backtracking and non-determinism

Input	Output
`[s,h,e,e,p]`	`e`
`[a,l,p,a,c,a]`	`a`
`[c,h,i,c,k,e,n]`	**c**

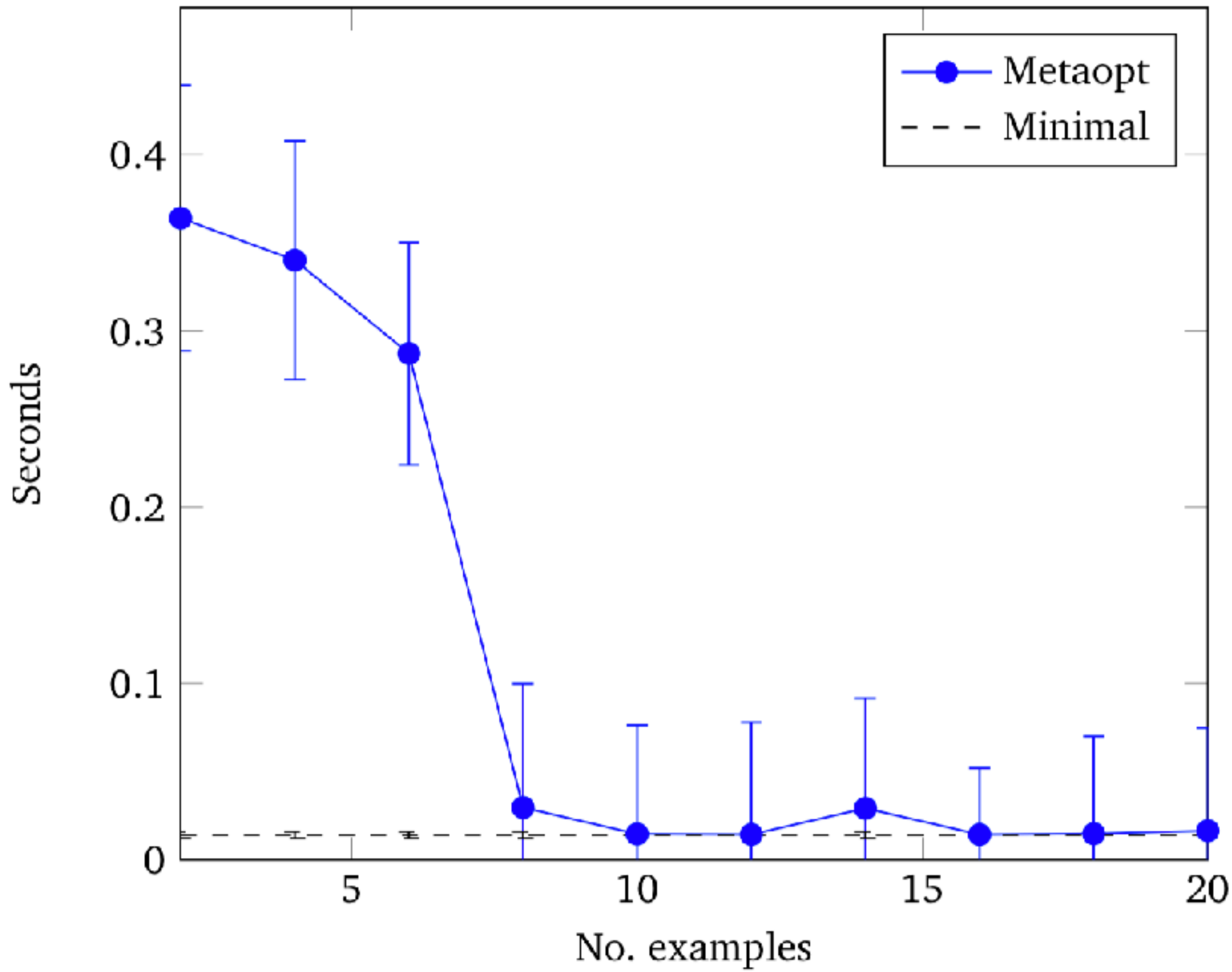
Input	Output
[s,h,e,e,p]	e
[a,l,p,a,c,a]	a
[c,h,i,c,k,e,n]	**c**

```
f(A,B):-mergesort(A,C),f1(C,B).
f1(A,B):-head(A,B),tail(A,C),head(C,B).
f1(A,B):-tail(A,C),f1(C,B).
```

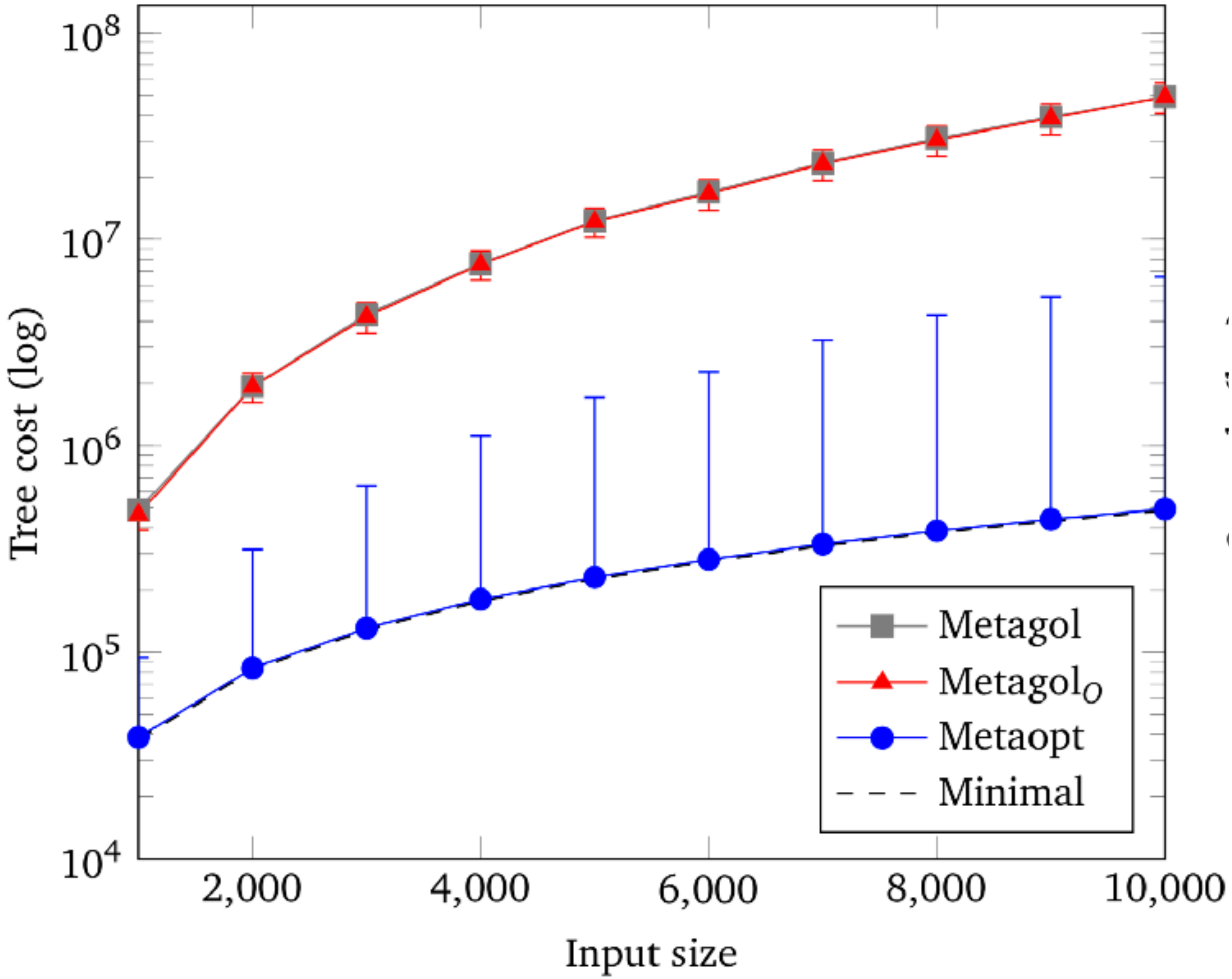
Convergence: program tree costs



Convergence: program running times

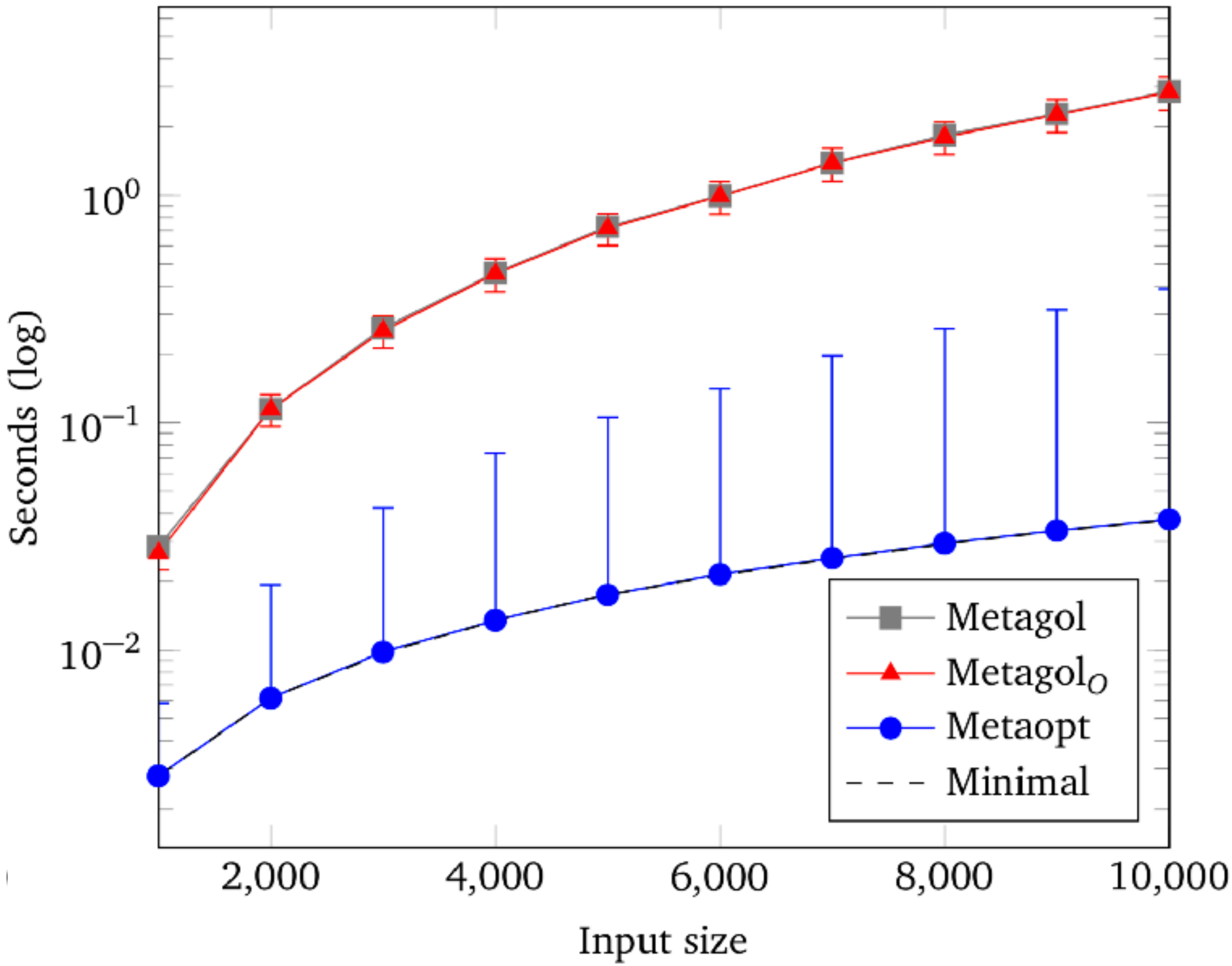


Performance



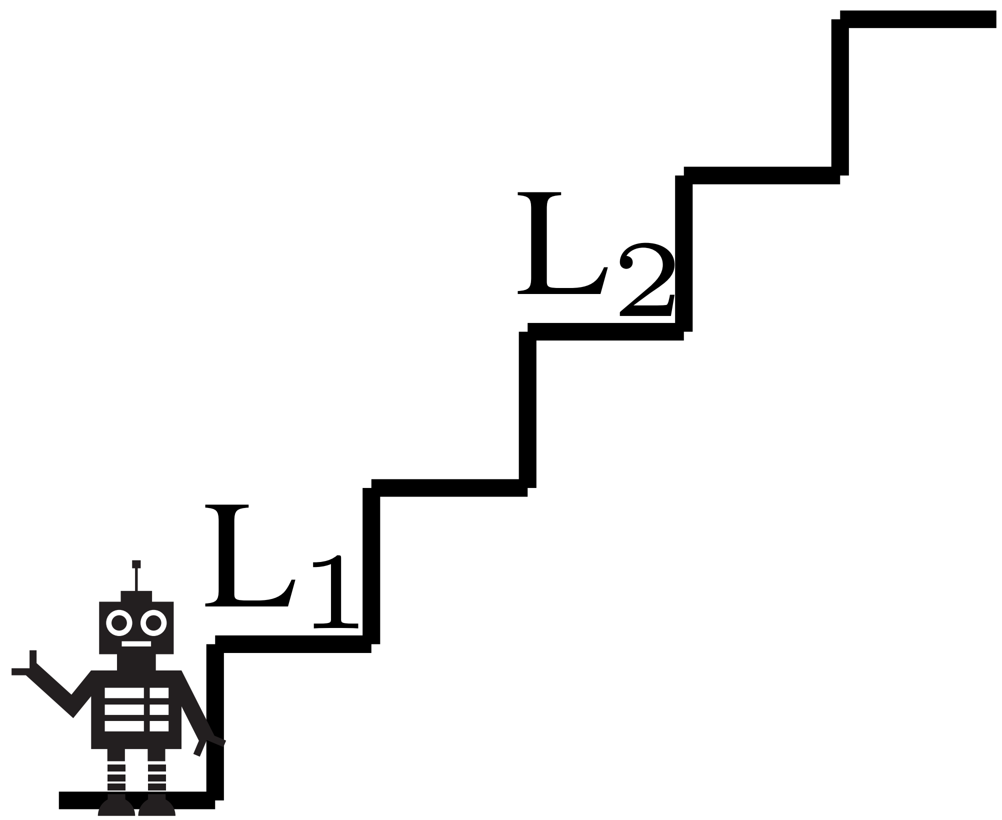
(a) Tree costs

Performance

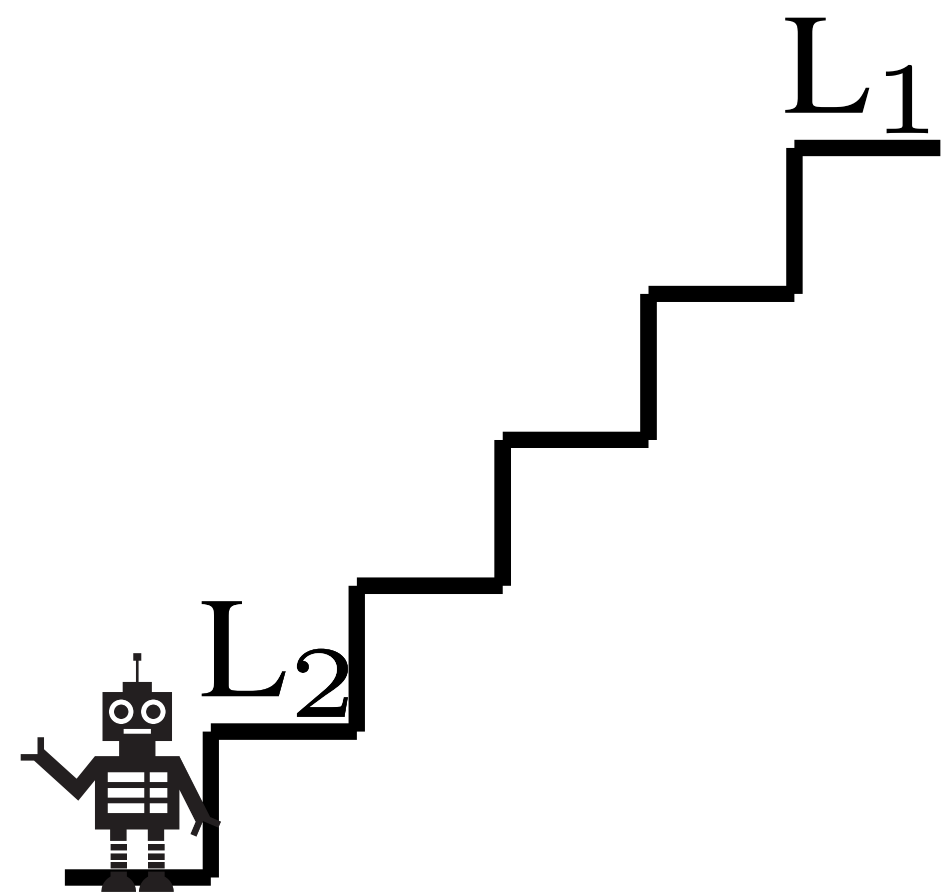


(b) Program runtimes

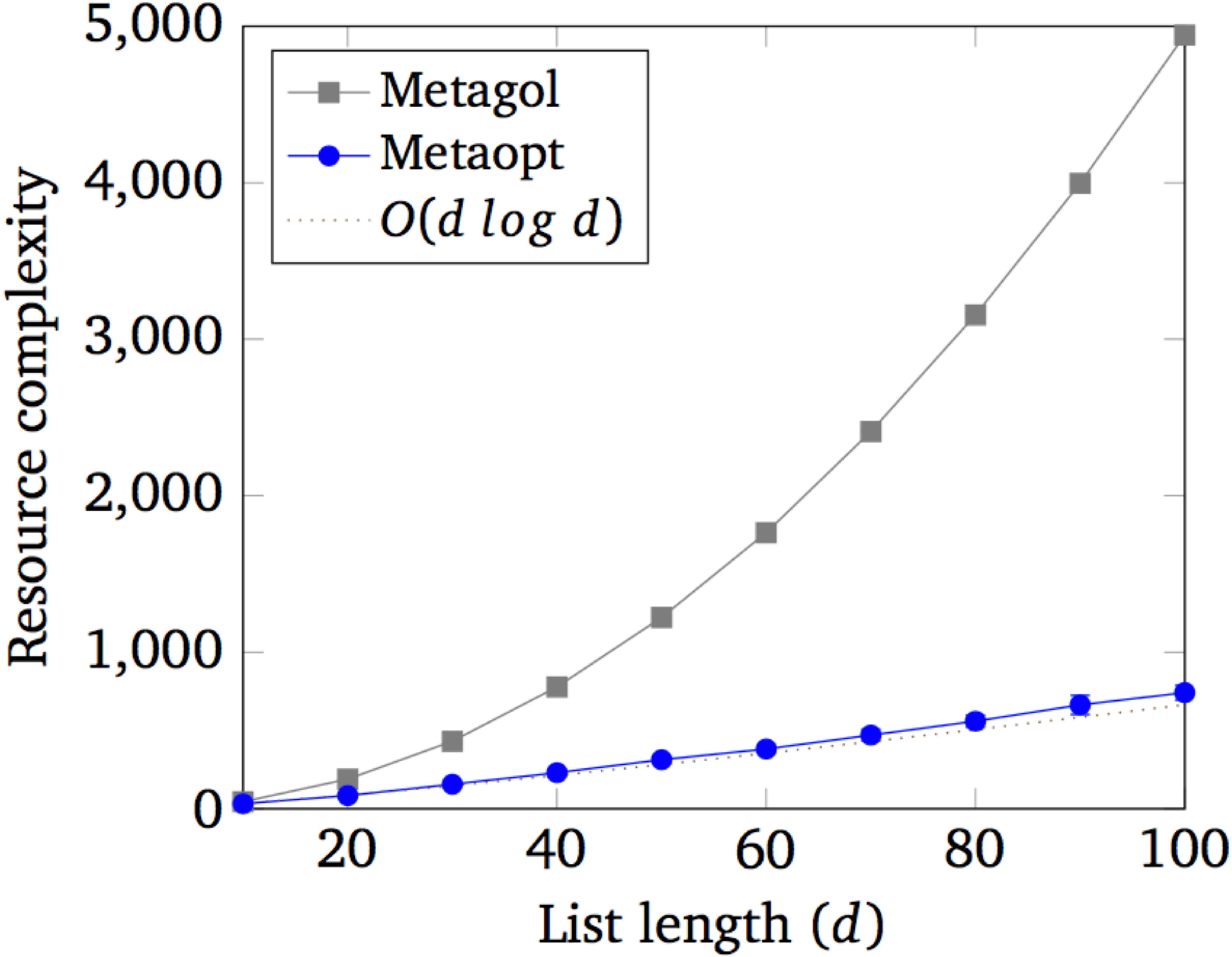
Resource complexity



Initial state

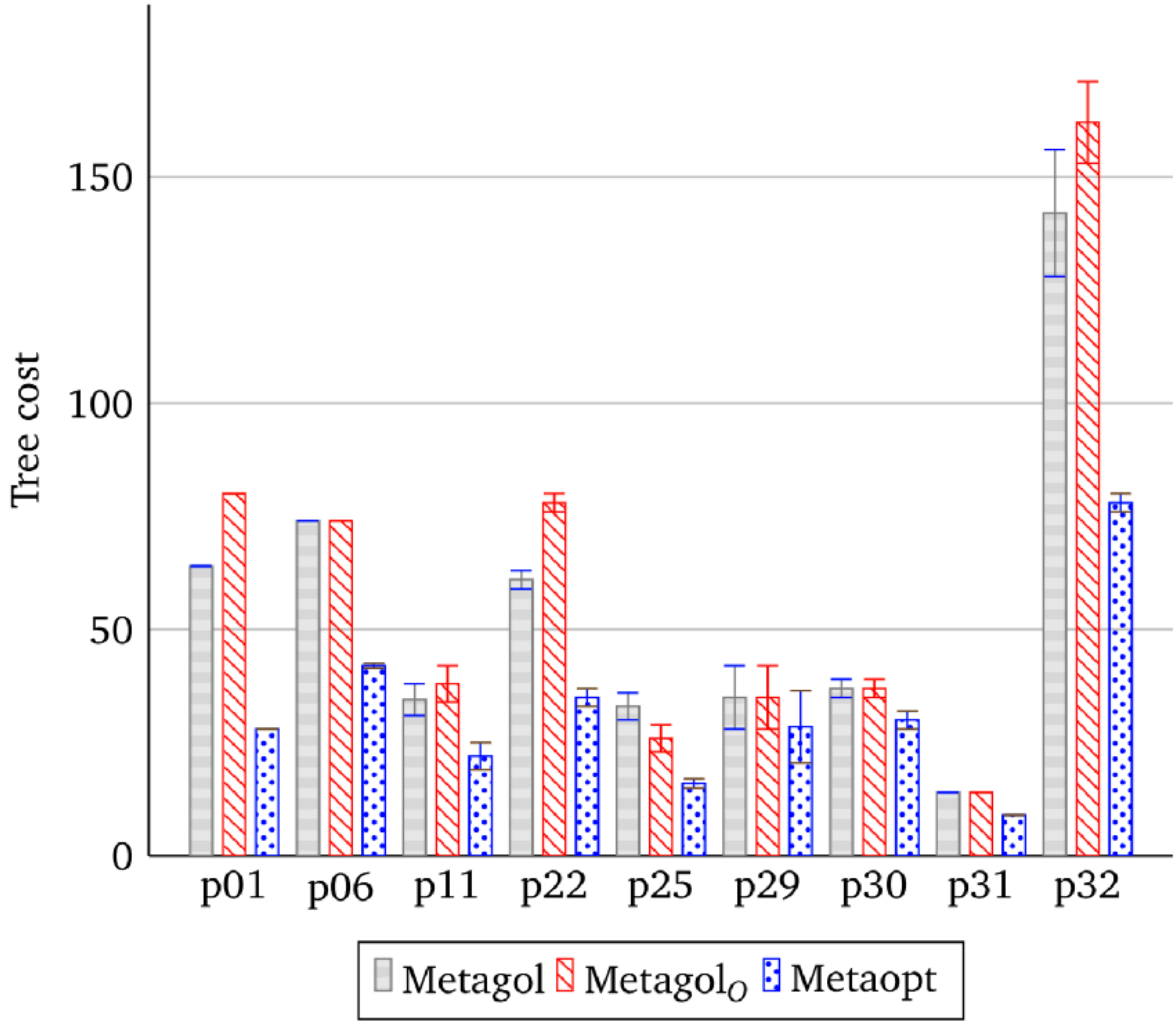


Final state



5,000
4,000
3,000
2,000
1,000
0
Resource complexity
Metagol
Metaopt
$O(d \log d)$
20
40
60
80
100
List length (d)

Input	Output
My name is John.	John
My name is Bill.	Bill
My name is Josh.	Josh
My name is Albert.	Albert
My name is Richard.	Richard



Tree cost
0
50
100
150
p01
p06
p11
p22
p25
p29
p30
p31
p32
Metagol
Metagol$_O$
Metaopt

```
%% metagol
f(A,B):-tail(A,C),f1(C,B).
f1(A,B):-dropLast(A,C),f2(C,B).
f2(A,B):-dropWhile(A,B,not_uppercase).
```

```
%% metagol unfolded
f(A,B):-
    tail(A,C),
    dropLast(C,D),
    dropWhile(D,B,not_uppercase).
```

```
%% metagol unfolded
f(A,B):-
    tail(A,C),
    dropLast(C,D),
    dropWhile(D,B,not_uppercase).
```

1

n-1

4n

```
% metagol0
f(A,B):-f1(A,C),f4(C,B).
f1(A,B):-f2(A,C),f3(C,B).
f2(A,B):-filter(A,B,is_letter).
f3(A,B):-dropWhile(A,B,is_uppercase).
f4(A,B):-dropWhile(A,B,not_uppercase).
```

```
% metagol0 unfolded
f(A,B):-
    filter(A,C,is_letter).
    dropWhile(C,D,is_uppercase),
    dropWhile(D,B,not_uppercase).
```

```
% metagol0 unfolded
f(A,B):-
    filter(A,C,is_letter).
    dropWhile(C,D,is_uppercase),
    dropWhile(D,B,not_uppercase).
```

4n

4n

4n

```
% learning f/2
% clauses: 1
% clauses: 2
% clauses: 3
% is better: 67
% is better: 57
% clauses: 4
% is better: 55
% clauses: 5
% is better: 53
% is better: 51
% is better: 49
% is better: 46
% clauses: 6
% is better: 41
% is better: 36
% is better: 31
f(A,B):-tail(A,C),f_1(C,B).
f_1(A,B):-f_2(A,C),dropLast(C,B).
f_2(A,B):-f_3(A,C),f_3(C,B).
f_3(A,B):-tail(A,C),f_4(C,B).
f_4(A,B):-f_5(A,C),f_5(C,B).
f_5(A,B):-tail(A,C),tail(C,B).
```

```
% metaopt
f(A,B):-tail(A,C),f1(C,B).
f1(A,B):-f2(A,C),dropLast(C,B).
f2(A,B):-f3(A,C),f3(C,B).
f3(A,B):-tail(A,C),f4(C,B).
f4(A,B):-f5(A,C),f5(C,B).
f5(A,B):-tail(A,C),tail(C,B).
```

```
% metaopt unfolded
f(A,B):-
    tail(A,C),
    tail(C,D),
    tail(D,E),
    tail(E,F),
    tail(F,G),
    tail(G,H),
    tail(H,I),
    tail(I,J),
    tail(J,K),
    tail(K,L),
    tail(L,M),
    dropLast(M,B).
```

```
% metaopt unfolded
f(A,B):-
    tail(A,C),
    tail(C,D),
    tail(D,E),
    tail(E,F),
    tail(F,G),
    tail(G,H),
    tail(H,I),
    tail(I,J),
    tail(J,K),
    tail(K,L),
    tail(L,M),
    dropLast(M,B).
```

does this last

todo

- Study complexity of Metaopt variants
- Characterise complexity of learned programs
- Learning efficient functional programs
- Discover new efficient algorithms

Todo (in general)

- Learning efficient programs
- Learning reusable higher-order abstractions
- Learning from minimal biases
- **Never-ending / lifelong learning**

Learning abstractions

Input	**Output**
[dagstuhl,2017]	[dagstuh,201]
[alice,bob,charlie]	[alic,bo,charli]
[1234,12,564]	[123,1,56]
[ab,abc,abcd,abcde]	???

Input	**Output**
[dagstuhl,2017]	[dagstuh,201]
[alice,bob,charlie]	[alic,bo,charli]
[1234,12,564]	[123,1,56]
[ab,abc,abcd,abcde]	**[a,ab,abc,abcd]**

```
f(A,B):-map(A,B,f1).
f1(A,B):-f2(A,C),tail(C,D),f2(D,B).
f2(A,B):-reduceback(A,B,concat).
```

Input	**Output**
[dagstuhl,2017]	[dagstuh]
[alice,bob,charlie]	[alic,bo]
[1234,12,564]	[123,1]
[ab,abc,abcd,abcde]	???

Input	Output
[dagstuhl,2017]	[dagstuh]
[alice,bob,charlie]	[alic,bo]
[1234,12,564]	[123,1]
[ab,abc,abcd,abcde]	**[a,ab,abc]**

```
f(A,B):-map(A,C,f1),f1(C,B).
f1(A,B):-f2(A,C),tail(C,D),f2(D,B).
f2(A,B):-reduceback(A,B,concat).
```

learn

```
f(A,B):-map(A,C,f1),f1(C,B).
f1(A,B):-f2(A,C),tail(C,D),f2(D,B).
f2(A,B):-reduceback(A,B,concat).
```

learn