

Learning efficient logic programs

Andrew Cropper & Stephen Muggleton

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

```
%% metagol
```

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

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

```
%% 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).
```

Idea

Input

- examples **E**
- background knowledge **B**
- **cost** : Program \times Example \rightarrow N

Idea

1. Learn any program **H**
2. Repeat while possible:
 - a. Learn program **H'** where $\mathbf{max_cost(H',E) < max_cost(H,E)}$
 - b. **H=H'**
3. Return **H**

Metagol

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

```
prove([Atom|Atoms], P1, P2):-  
    prove_aux(Atom, P1, P3),  
    prove(Aatoms, 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(Aatoms, 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(Aatoms, 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(Aatoms,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(Aatoms,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(Aatoms,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 $\mathbf{max_cost(H',E) < max_cost(H,E)}$
 - 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

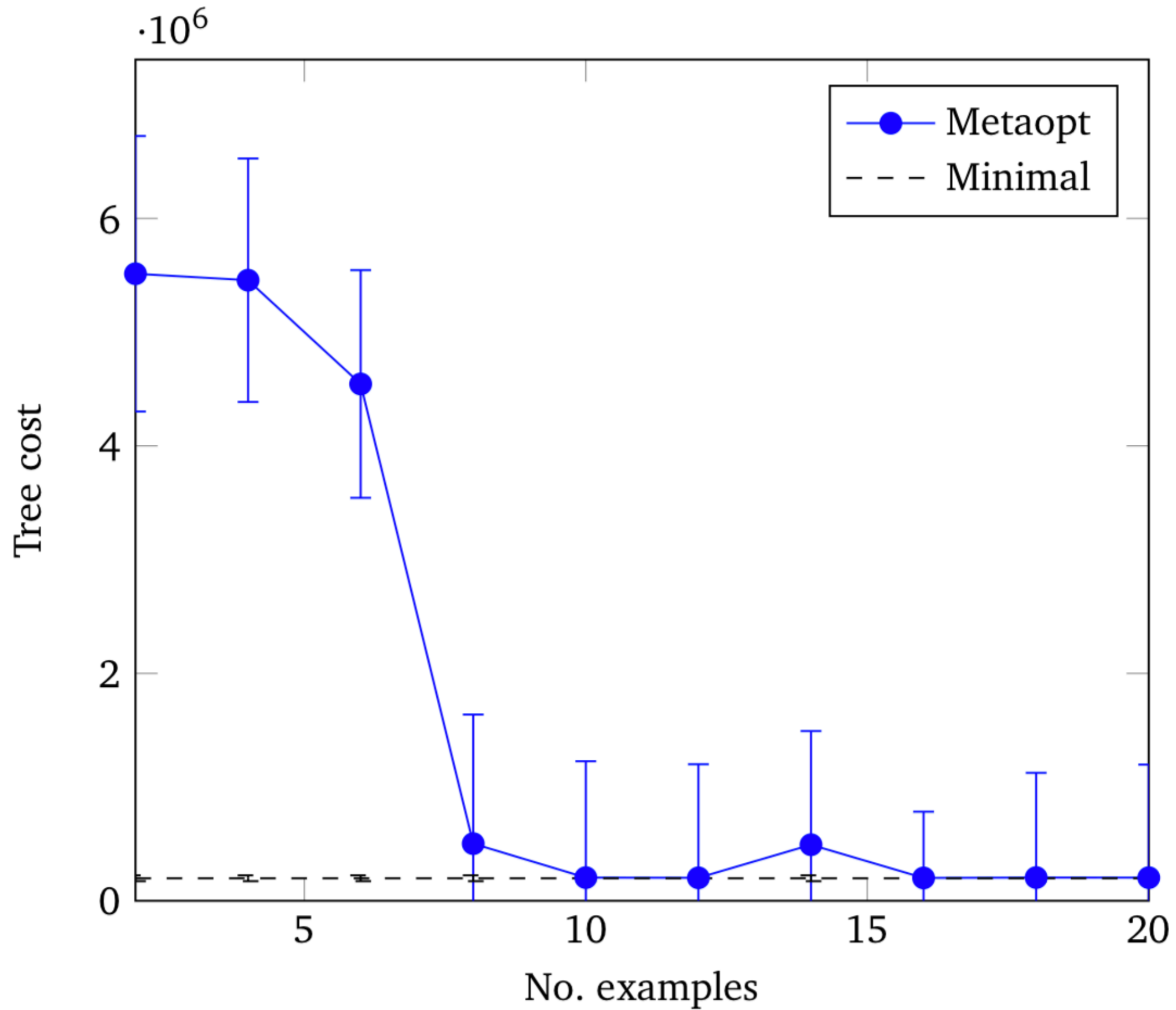
Input	Output
[s,h,e,e,p]	e
[a,l,p,a,c,a]	a
[c,h,i,c,k,e,n]	c

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

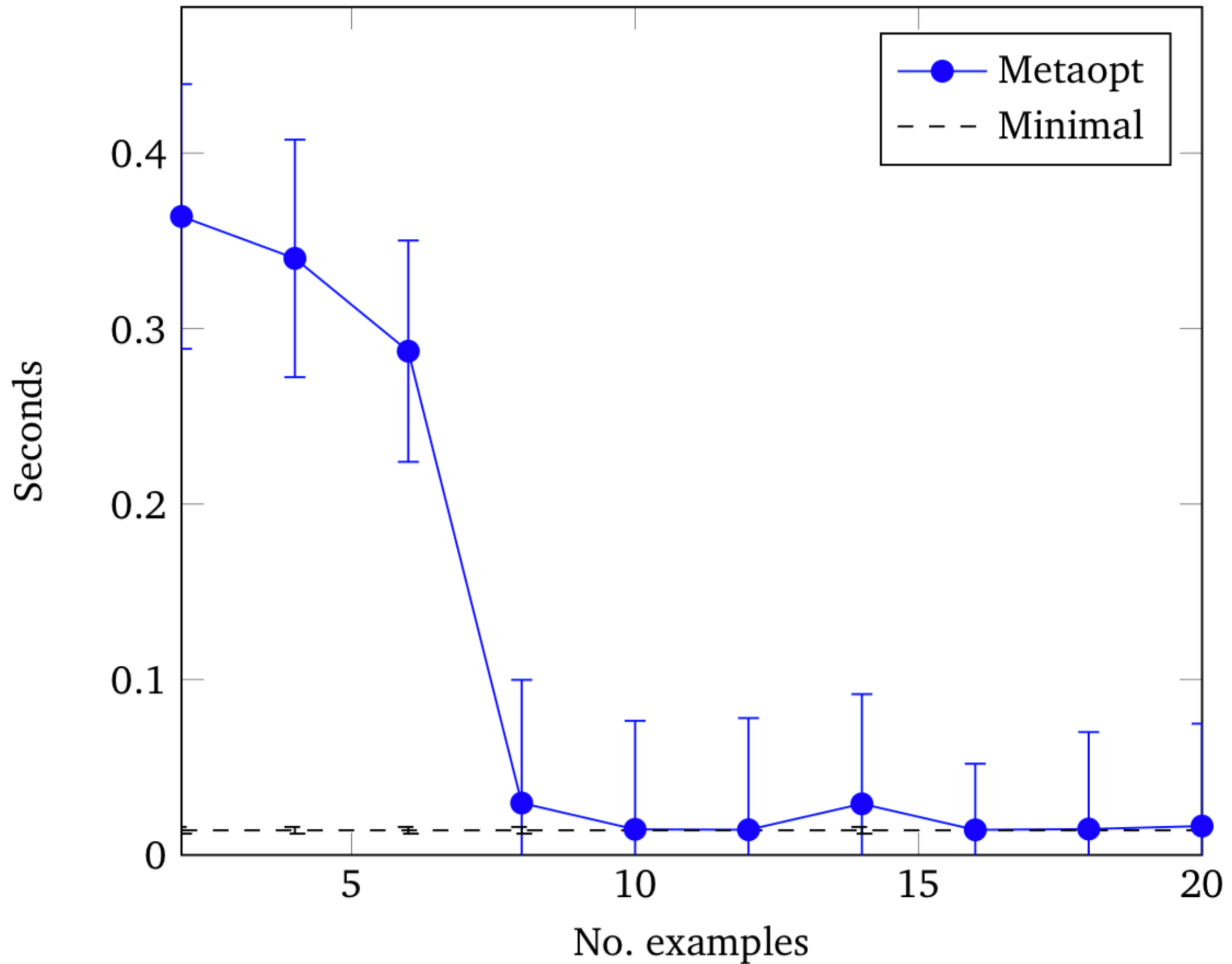
`f1(A,B):-head(A,B),tail(A,C),head(C,B).`

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

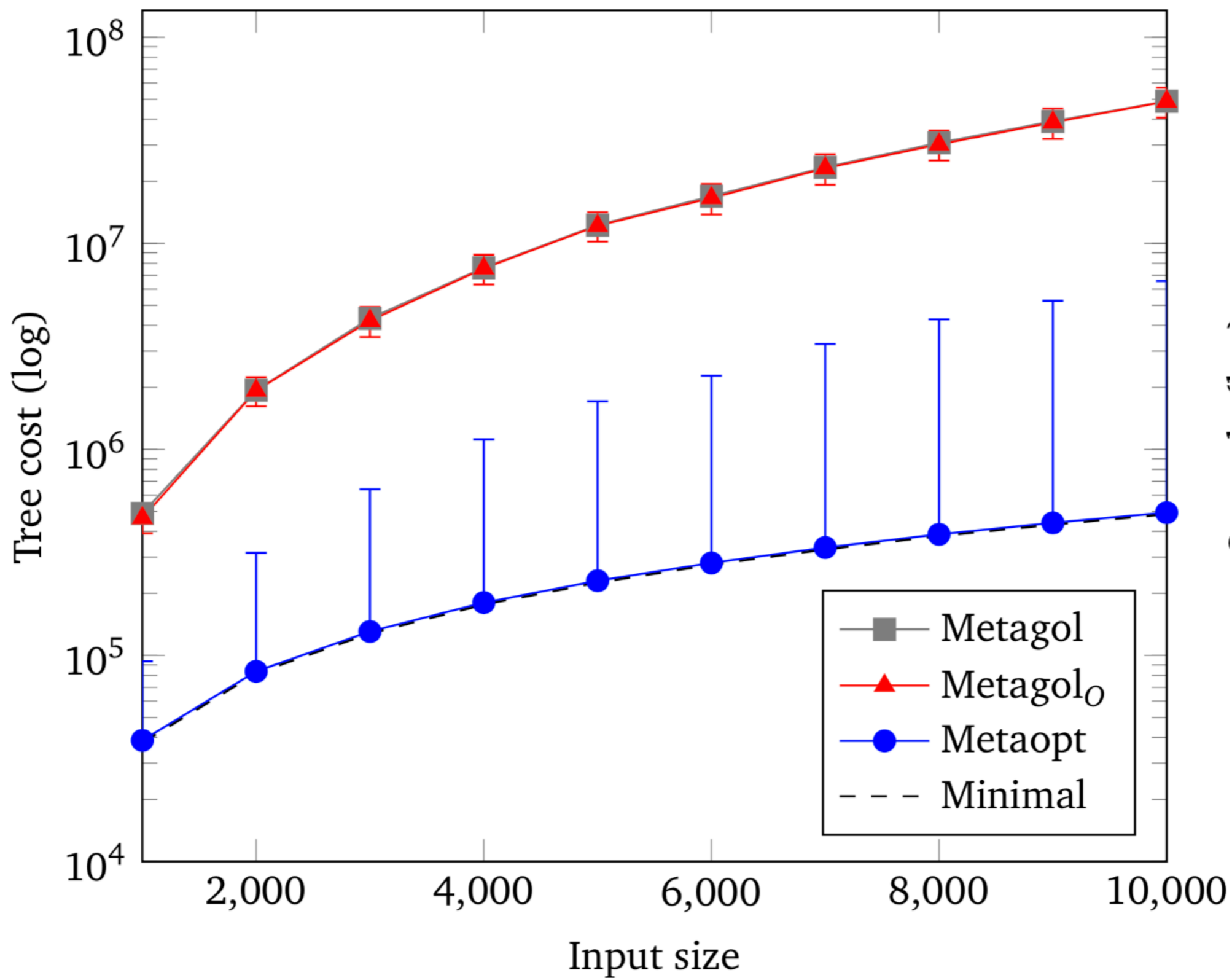
Convergence: program tree costs



Convergence: program running times

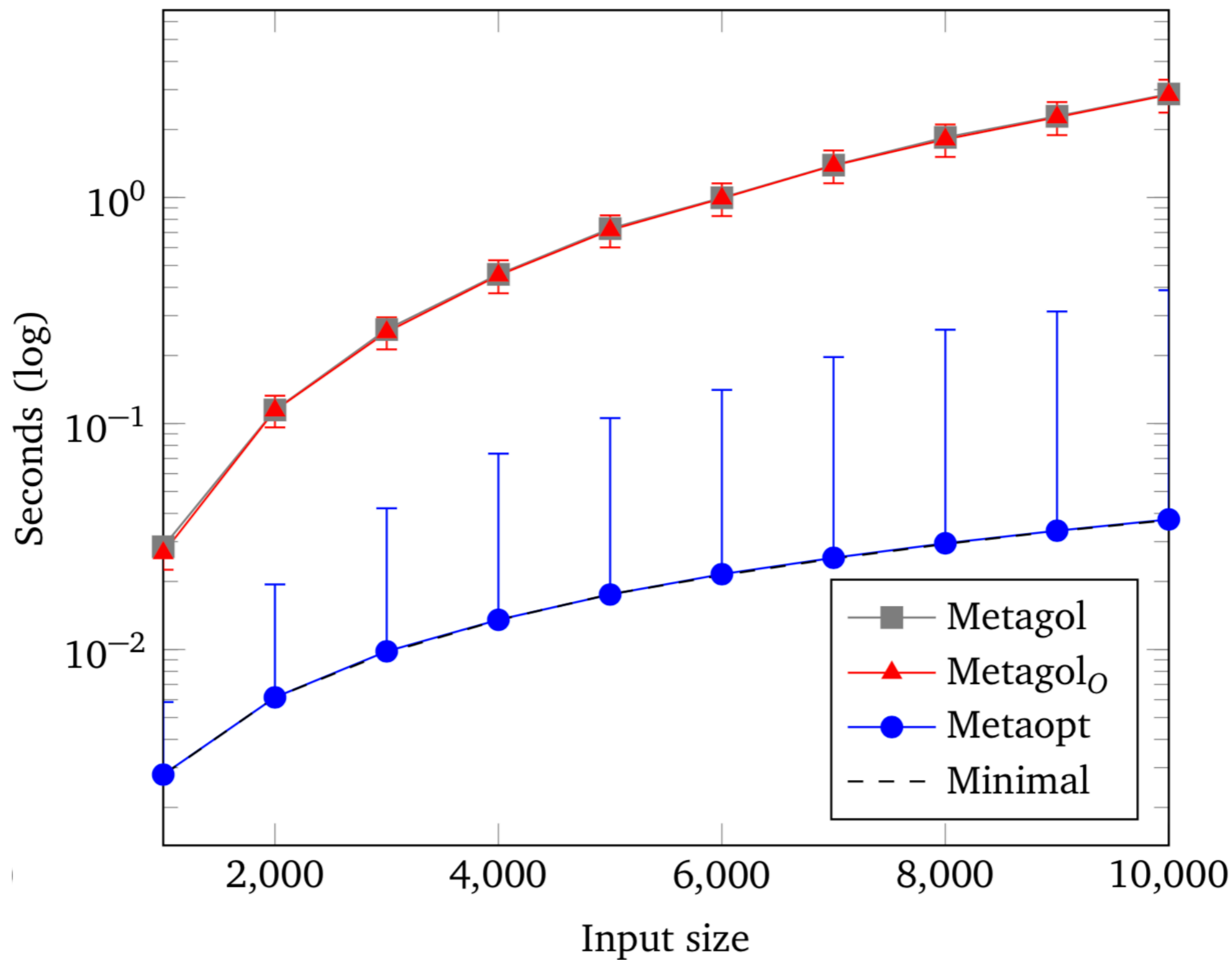


Performance



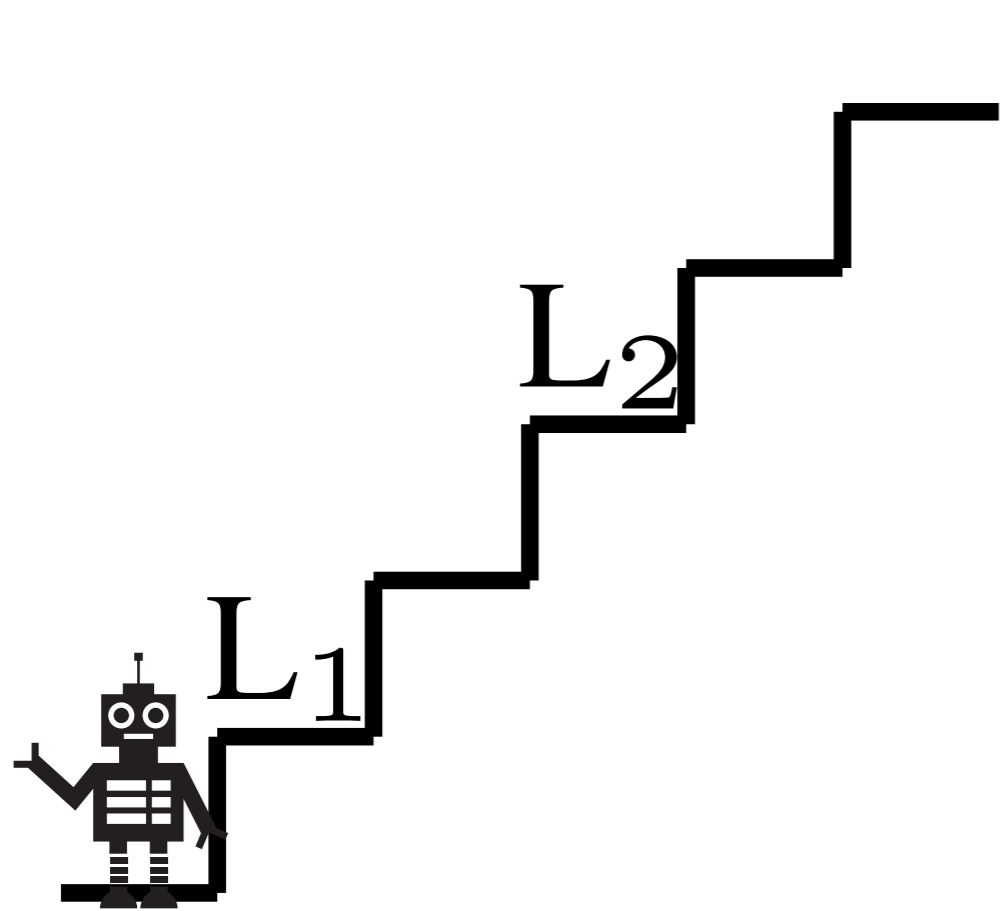
(a) Tree costs

Performance

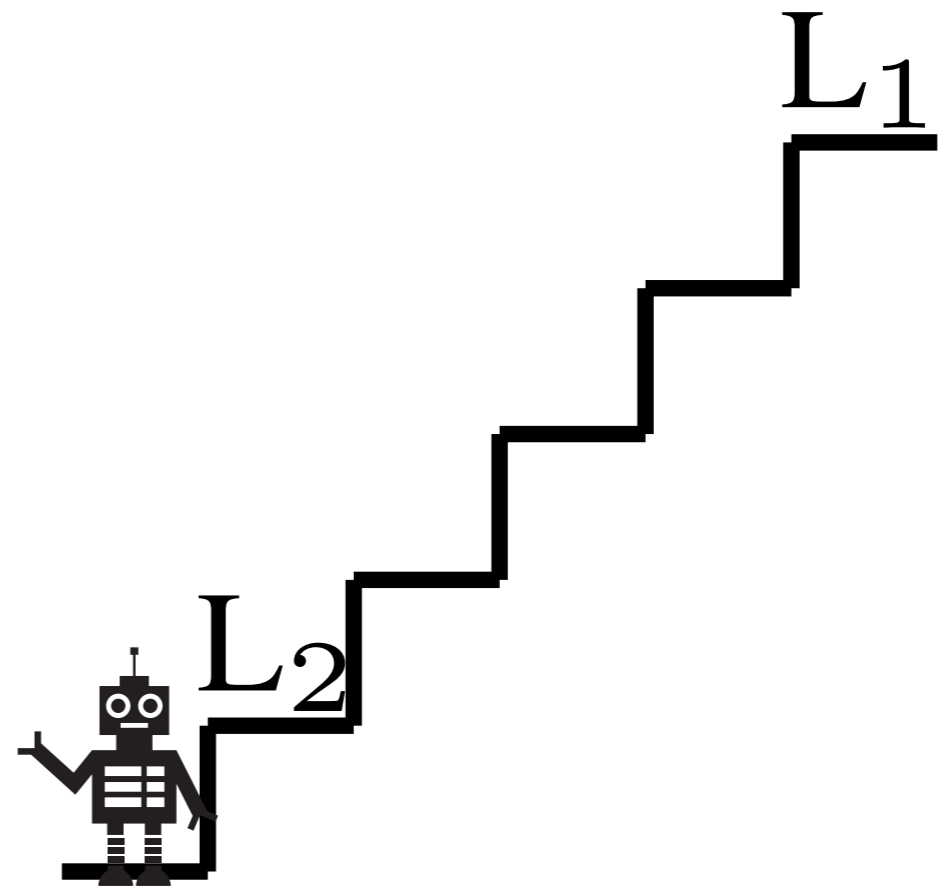


(b) Program runtimes

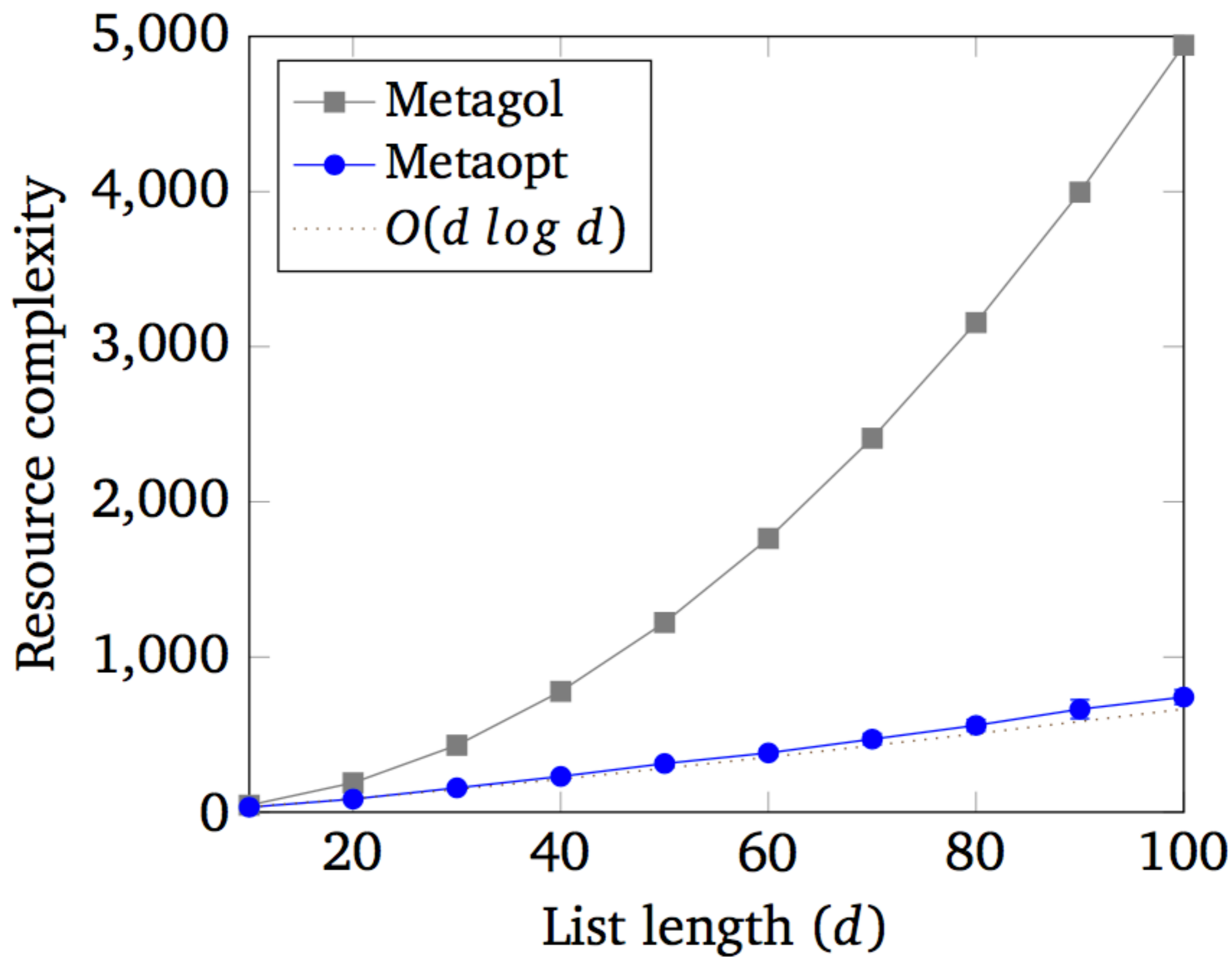
Resource complexity



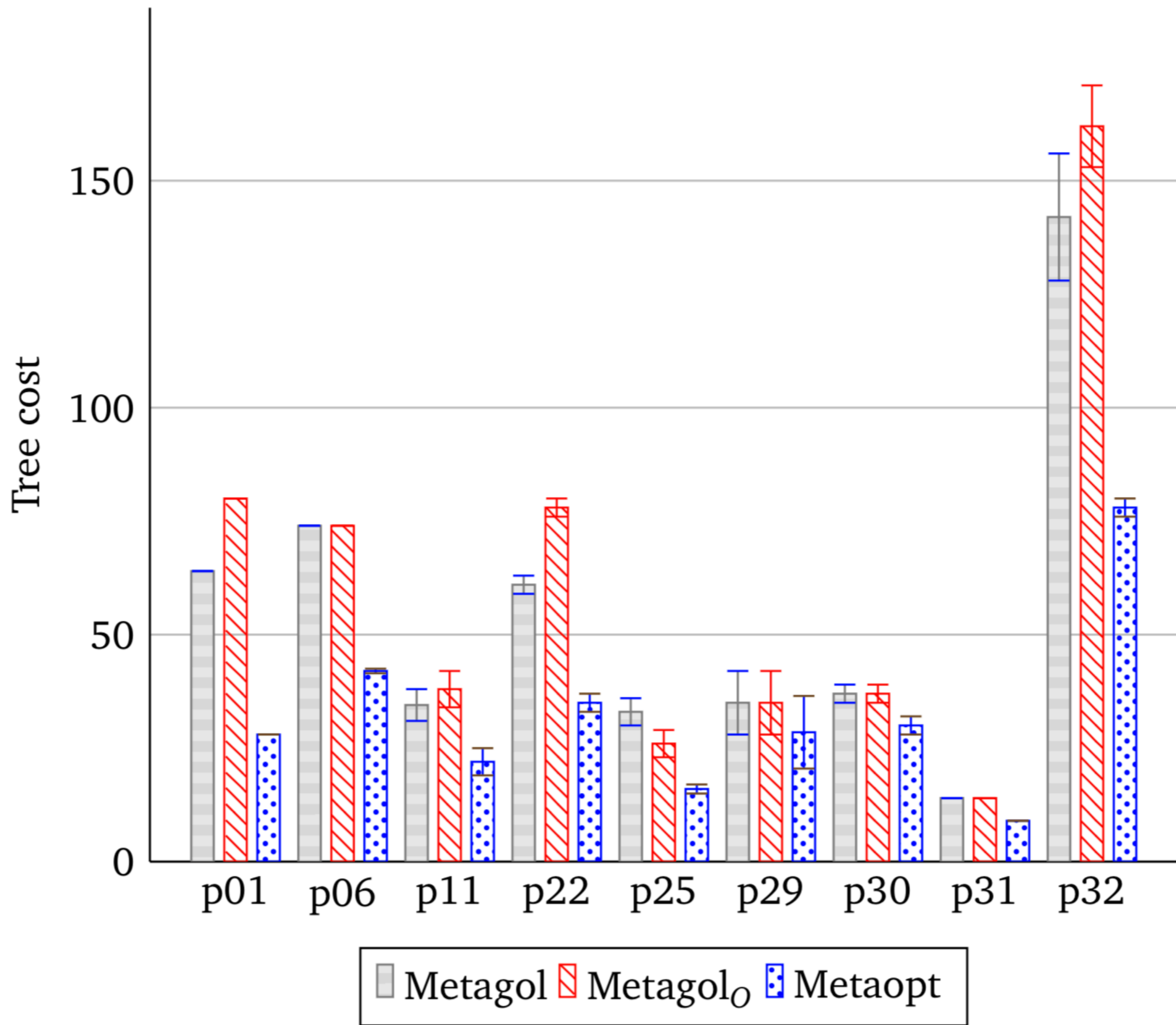
Initial state



Final state



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



```
%% 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).
```

```
% 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).
```

```
% 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
- Discover new efficient algorithms