EECS 598-008 & EECS 498-008: Intelligent Programming Systems

Lecture 7

Announcements

- CFPP out, due **midnight Tuesday September 28**
  - How paper assignment, review, presentation work
  - 0 points in final grade but very important
  - Create HotCRP account
  - **Submit your paper presentation preferences**
- **Live**, remote discussion 3-4pm Friday September 24
  - Zoom link on course website
  - Discuss A2 (due midnight September 27)
- Course survey (sent in Slack channel)
  - Used to improve the course
  - Everyone gets 1 extra point if >80% of the class take it

Today’s Agenda

- **Pruning (review)**
- Search prioritization

Previous Lecture

- SMT-based deduction to prune partial programs in top-down search
- Goal: speed up search
  - Prune space of programs without exploring the space

Previous Lecture

- SMT-based deduction to prune partial programs in top-down search
- Goal: speed up search
  - Prune space of programs without exploring the space
- Idea: given a partial program, encode it as an SMT formula, check SAT
  - If SAT: cannot prune away partial program
  - If UNSAT: no need to explore concrete programs derived from partial program

SMT-based Pruning

- Step 1: define abstract semantics of each DSL operator in SMT

SMT-based Pruning

- Step 1: define abstract semantics of each DSL operator in SMT
- Step 2: given partial program P, generate an SMT formula ϕ_P that encodes abstract behavior of P

SMT-based Pruning

- Step 1: define abstract semantics of each DSL operator in SMT
- Step 2: given partial program P, generate an SMT formula ϕ_P that encodes abstract behavior of P
- Step 3: encode input-output example as SMT formula ϕ_E

SMT-based Pruning

- Step 1: define abstract semantics of each DSL operator in SMT
- Step 2: given partial program P, generate an SMT formula ϕ_P that encodes abstract behavior of P
- Step 3: encode input-output example as SMT formula ϕ_E
- Step 4: SAT($\phi_P \wedge \phi_E$)

SMT-based Pruning

- Step 1: define abstract semantics of each DSL operator in SMT

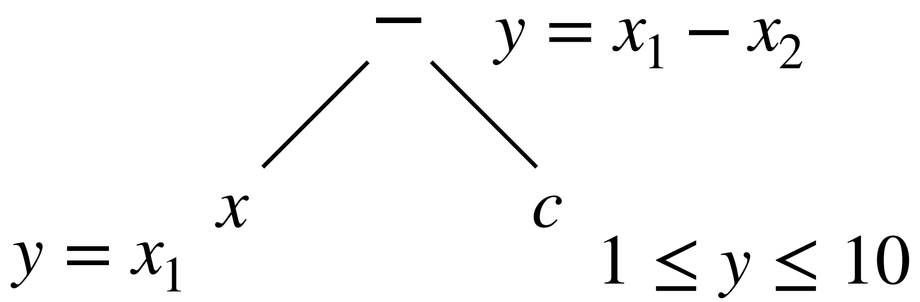
$$e ::= x \mid c \mid e + e \mid e - e \qquad c ::= 1 \mid 2 \mid \cdots \mid 10$$

$y = x_1$ $\quad 1 \le y \le 10$ $\quad y = x_1 + x_2$ $\quad y = x_1 - x_2$

Example: 5 —> 20

SMT-based Pruning

- Step 2: given partial program P, generate an SMT formula ϕ_P that encodes abstract behavior of P



$$\phi_P : (y_1 = x) \wedge (1 \leq y_2 \leq 10) \wedge (y = y_1 - y_2)$$

SMT-based Pruning

- Step 3: encode input-output example as SMT formula ϕ_E

$$\phi_E : (x = 5) \wedge (y = 20)$$

SMT-based Pruning

- Step 4: SAT($\phi_P \wedge \phi_E$)

$$\phi_P \wedge \phi_E : (y_1 = x) \wedge (1 \leq y_2 \leq 10) \wedge (y = y_1 - y_2) \wedge (x = 5) \wedge (y = 20)$$

SMT-based Pruning

- Step 4: SAT($\phi_P \wedge \phi_E$)

$$\phi_P \wedge \phi_E : (y_1 = x) \wedge (1 \leq y_2 \leq 10) \wedge (y = y_1 - y_2) \wedge (x = 5) \wedge (y = 20)$$

UNSAT

Top-Down Search Algorithm with Deduction-based Pruning

Top-Down-Search ($(T, N, P, S), E$):

worklist := { S };

while (*worklist* **is not empty**):

AST := *worklist*.remove();

if (AST **is complete &** AST **satisfies** E): **return** AST;

if (**prune**(AST)): **continue**; // **Pruning**

worklist.addAll(**expand**(AST));

Top-Down Search Algorithm with Deduction-based Pruning

Top-Down-Search ($(T, N, P, S), E$):

worklist := { S };

while (*worklist* **is not empty**):

AST := *worklist*.remove();

if (AST **is complete &** AST **satisfies** E): **return** AST;

if (**prune**(AST)): **continue**; // **Pruning**

worklist.addAll(**expand**(AST));

- Pruning is **sound**: if pruned, it means there is no concrete program that can be derived from the partial program and that satisfies the spec — **we don't miss anything**

Top-Down Search Algorithm with Deduction-based Pruning

Top-Down-Search ($(T, N, P, S), E$):

worklist := { S };

while (*worklist* **is not empty**):

AST := *worklist*.remove();

if (AST **is complete &** AST **satisfies** E): **return** AST;

if (**prune**(AST)): **continue**; // **Pruning**

worklist.addAll(**expand**(AST));

- Pruning is **sound**: if pruned, it means there is no concrete program that can be derived from the partial program and that satisfies the spec — **we don't miss anything**
- Pruning may be **incomplete**: if there is no program from the partial program P that satisfies the spec, we may not be able to prune P — **false alarms**

Today’s Agenda

- Pruning (review)
- **Search prioritization**

Pruning Alone is Not Sufficient

- So far, mostly about pruning
  - Which programs that we **provably** don't need to inspect?
  - Guarantee: not miss any programs that satisfy the spec

Pruning Alone is Not Sufficient

- So far, mostly about pruning
  - Which programs that we **provably** don’t need to inspect?
  - Guarantee: not miss any programs that satisfy the spec
  - Top-down: prune “infeasible" (partial) programs
  - Bottom-up: prune “redundant” programs

Pruning Alone is Not Sufficient

- So far, mostly about pruning
  - Which programs that we **provably** don't need to inspect?
  - Guarantee: not miss any programs that satisfy the spec
  - Top-down: prune "infeasible" (partial) programs
  - Bottom-up: prune "redundant" programs
- Oftentimes, pruning **alone** is not sufficient
  - In addition to pruning, **search order** is also important
  - **Prioritize** search!

Search Prioritization

- Search prioritization: in what order do we search programs?

Search Prioritization

- Search prioritization: in what order do we search programs?
- Why do we need search prioritization?
  - **Acceleration**: speed up search to find a program that satisfies the spec
  - **Generalization**: use fewer examples to find a program that satisfies user intent

Generalization

- Does the synthesized program generalize to **unseen** examples?

Generalization

- Does the synthesized program generalize to **unseen** examples?
- For instance, consider $\{2 \mapsto 4,\ 3 \mapsto 6\}$
  - What's the intended program?

Generalization

- Does the synthesized program generalize to **unseen** examples?
- For instance, consider $\{2 \mapsto 4,\ 3 \mapsto 6\}$
  - What's the intended program?
- Why do we need to generalize?
  - Because inductive specs are under-constrained
  - Not focus on today's lecture (but we'll talk about this next week)

Search Prioritization

- Search prioritization: in what order do we search programs?
- Why do we need search prioritization?
  - **Acceleration**: speed up search to find a program that satisfies the spec
  - **Generalization**: use fewer examples to find a program that satisfies user intent
- Acceleration and generalization are not always orthogonal to each other
  - Share techniques
  - Acceleration may help generalization and vice versa

Search Prioritization

- Search prioritization: in what order do we search programs?
- Why do we need search prioritization?
  - **Acceleration**: speed up search to find a program that satisfies the spec
  - **Generalization**: use fewer examples to find a program that satisfies user intent
- Acceleration and generalization are not always orthogonal to each other
  - Share techniques
  - Acceleration may help generalization and vice versa
- Today's lecture
  - Focus on acceleration
  - Don't differentiate acceleration and generalization

Idea 1: DFS-Style Search

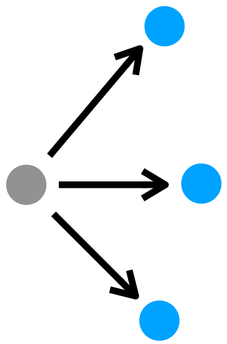
- Idea: exhaustively search “one branch”, then backtrack and go to another branch

Idea 1: DFS-Style Search

- Idea: exhaustively search “one branch”, then backtrack and go to another branch

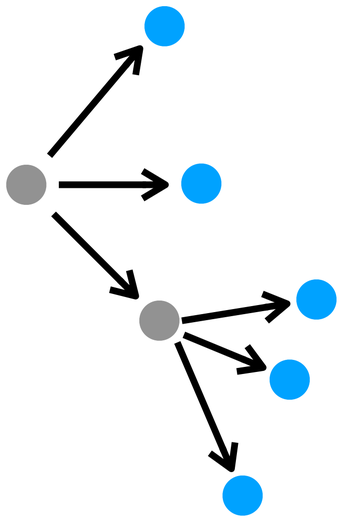
Idea 1: DFS-Style Search

- Idea: exhaustively search “one branch”, then backtrack and go to another branch



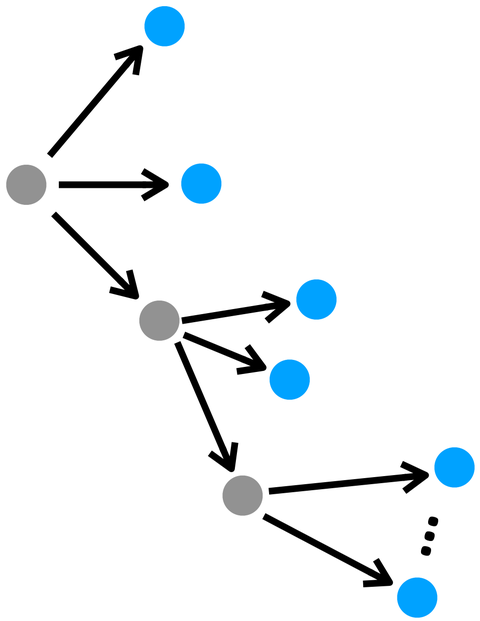
Idea 1: DFS-Style Search

- Idea: exhaustively search “one branch”, then backtrack and go to another branch



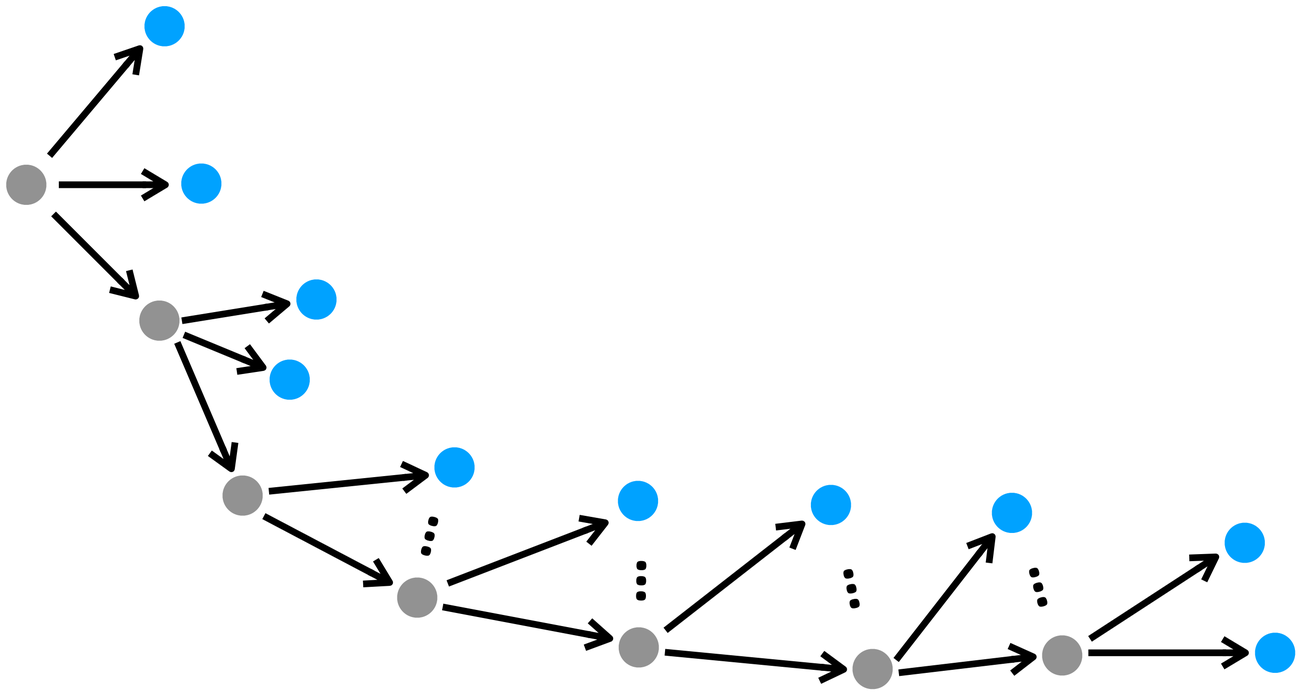
Idea 1: DFS-Style Search

- Idea: exhaustively search “one branch”, then backtrack and go to another branch



Idea 1: DFS-Style Search

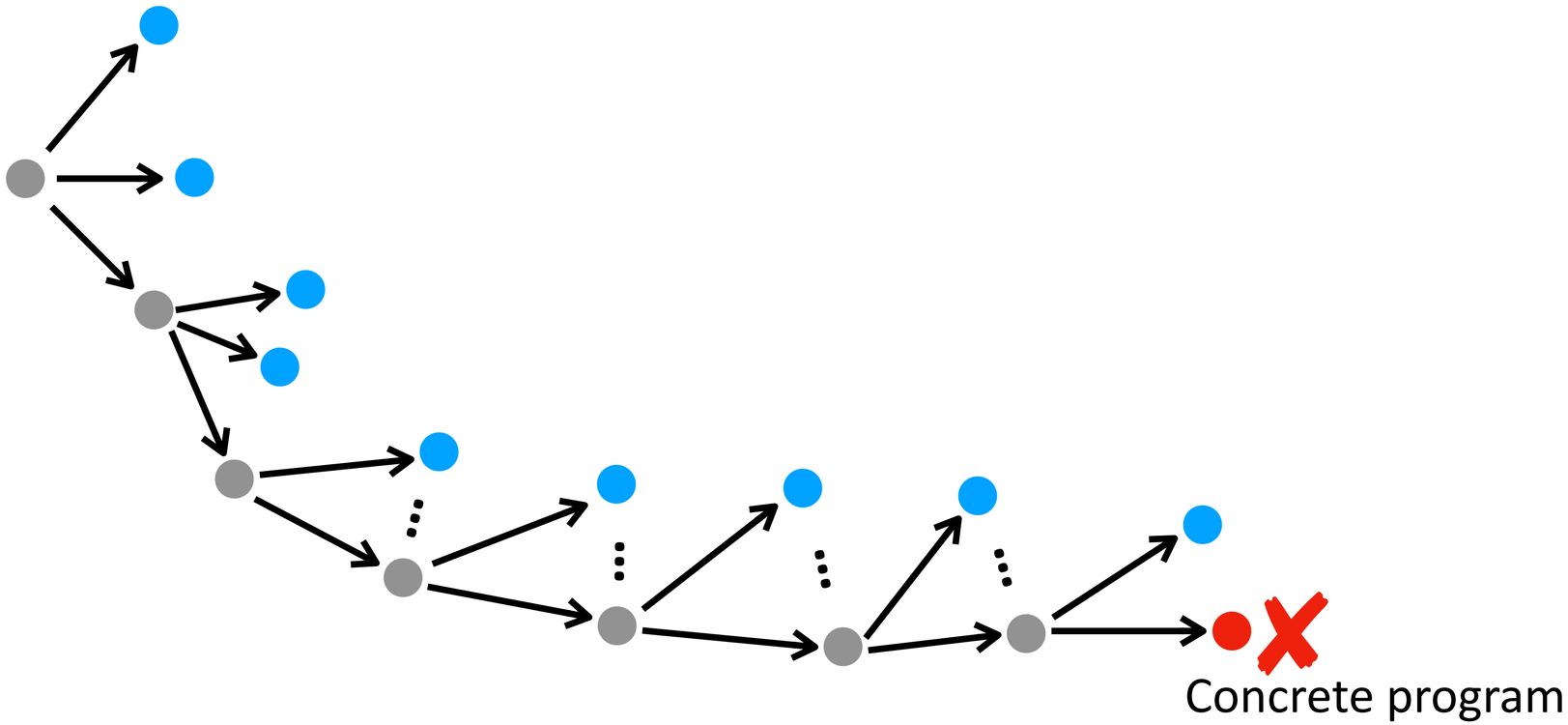
- Idea: exhaustively search “one branch”, then backtrack and go to another branch



Concrete program

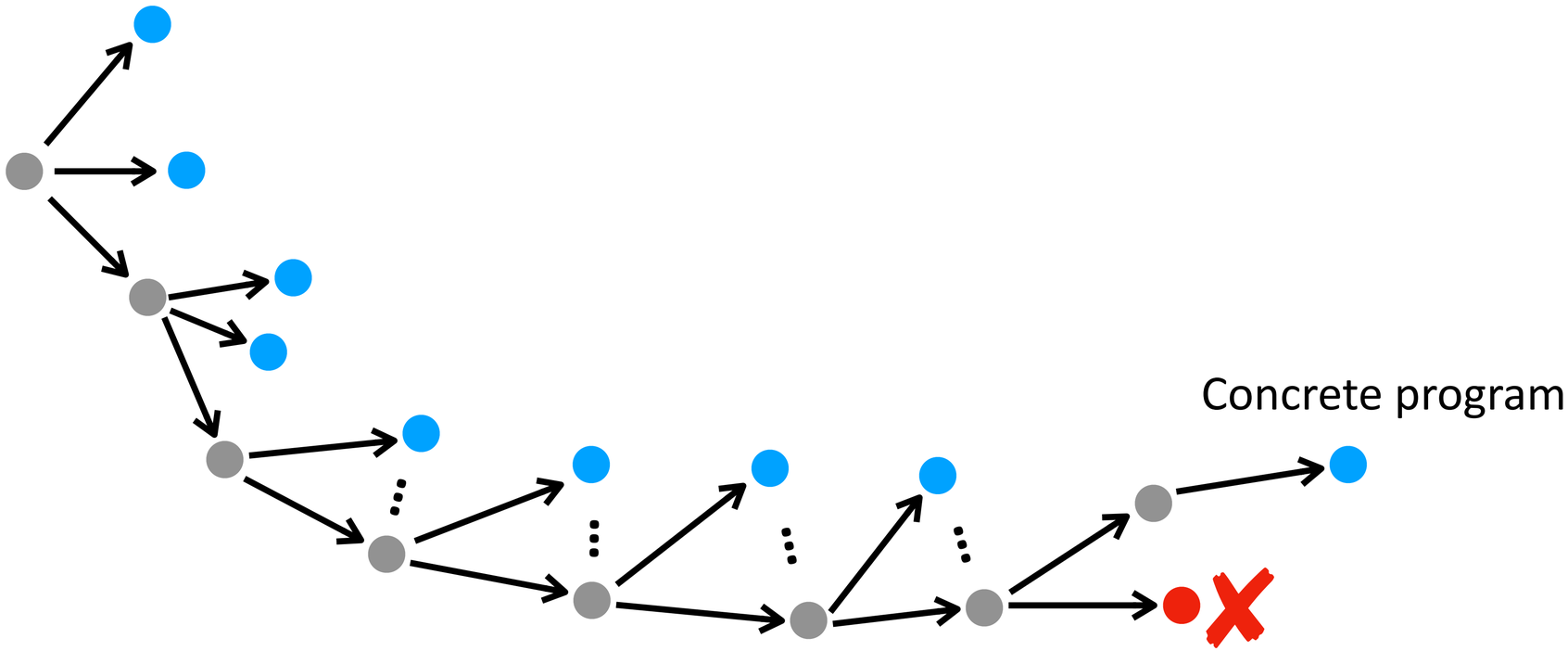
Idea 1: DFS-Style Search

- Idea: exhaustively search “one branch”, then backtrack and go to another branch



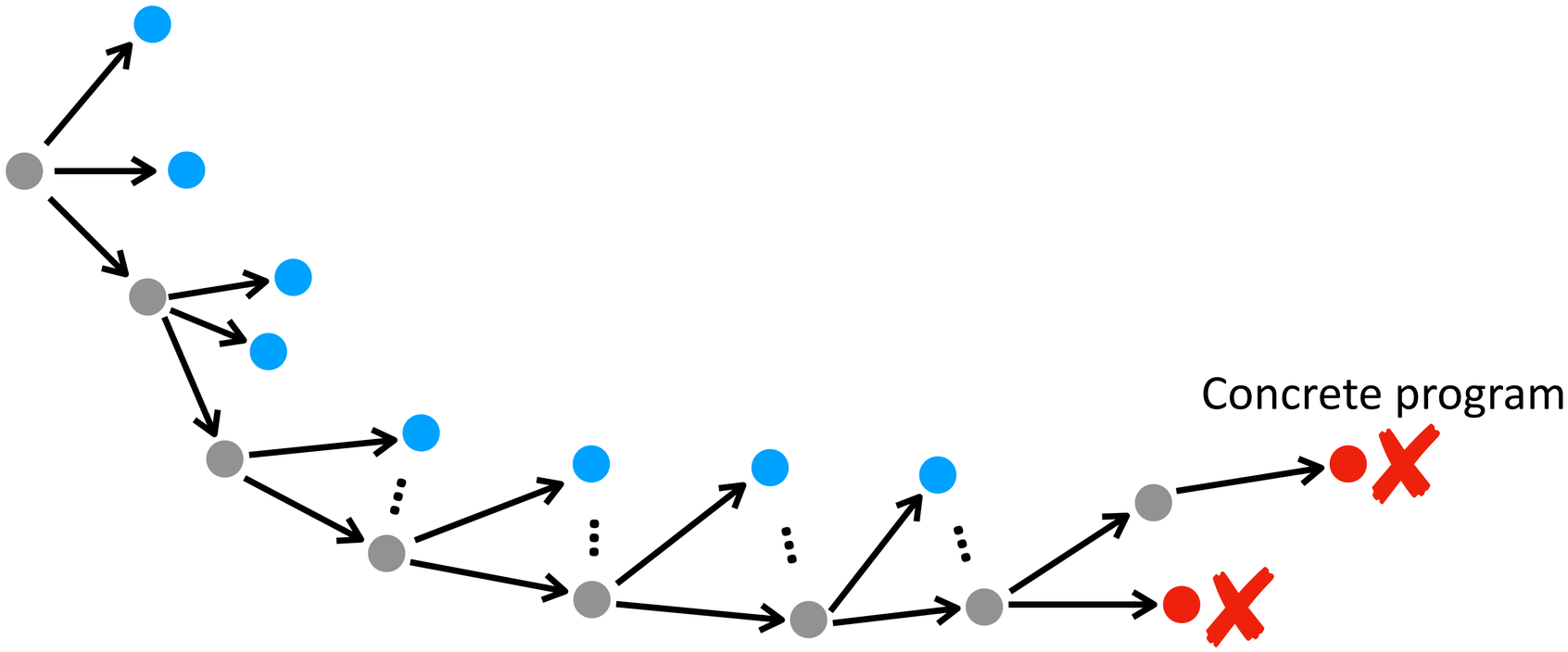
Idea 1: DFS-Style Search

- Idea: exhaustively search “one branch”, then backtrack and go to another branch



Idea 1: DFS-Style Search

- Idea: exhaustively search “one branch”, then backtrack and go to another branch



Idea 1: DFS-Style Search

- Idea: exhaustively search “one branch”, then backtrack and go to another branch
- Implementation

Top-Down-Search ($(T, N, P, S), E$):

worklist := { S }; // **Use a stack (first in last out)**

while (*worklist* **is not empty**):

AST := *worklist*.remove();

if (AST **is complete &** AST **satisfies** E): **return** AST;

worklist.addAll(**expand**(AST));

Idea 1: DFS-Style Search

- Idea: exhaustively search “one branch”, then backtrack and go to another branch
- Implementation
- Pros: simple, easy to implement

Idea 1: DFS-Style Search

- Idea: exhaustively search “one branch”, then backtrack and go to another branch
- Implementation
- Pros: simple, easy to implement
- What could go wrong?

Idea 1: DFS-Style Search

- Idea: exhaustively search “one branch”, then backtrack and go to another branch
- Implementation
- Pros: simple, easy to implement
- What could go wrong?
  - Have to bound “depth” of each branch; otherwise, not terminate

Idea 1: DFS-Style Search

- Idea: exhaustively search “one branch”, then backtrack and go to another branch
- Implementation
- Pros: simple, easy to implement
- What could go wrong?
  - Have to bound “depth” of each branch; otherwise, not terminate
  - May miss simpler solutions

Search Prioritization

- Idea 1: DFS-style
- Idea 2: BFS-style

Idea 2: BFS-Style Search

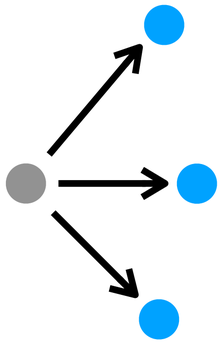
- Idea: simultaneously make progress on all branches

Idea 2: BFS-Style Search

- Idea: simultaneously make progress on all branches

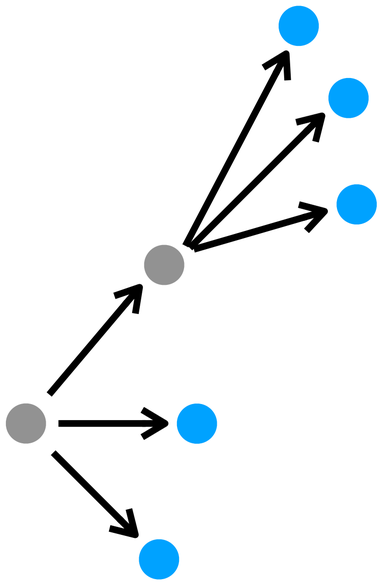
Idea 2: BFS-Style Search

- Idea: simultaneously make progress on all branches



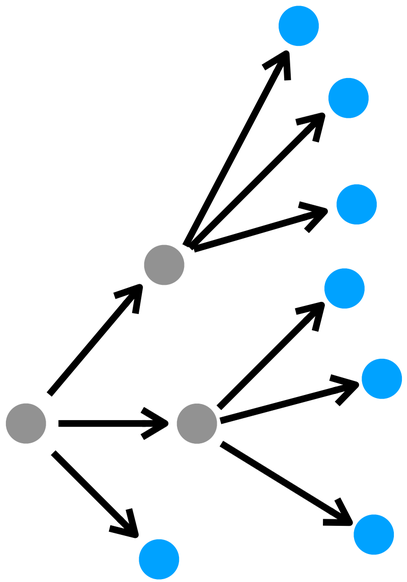
Idea 2: BFS-Style Search

- Idea: simultaneously make progress on all branches



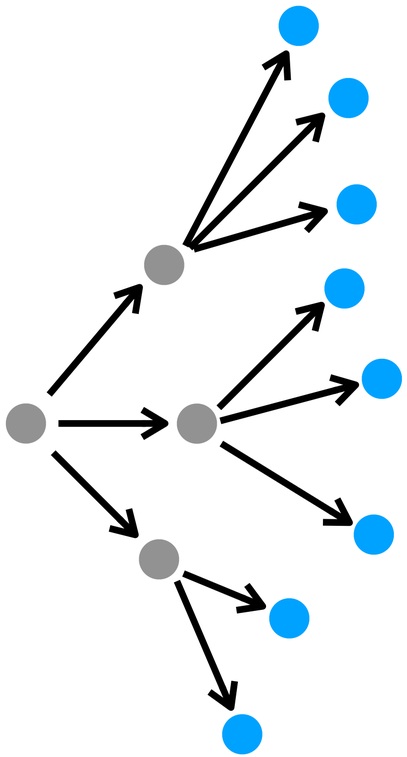
Idea 2: BFS-Style Search

- Idea: simultaneously make progress on all branches



Idea 2: BFS-Style Search

- Idea: simultaneously make progress on all branches



Idea 2: BFS-Style Search

- Idea: simultaneously make progress on all branches
- Implementation

Top-Down-Search ($(T, N, P, S), E$):

worklist := { S }; // **Use a queue (first in first out)**

while (*worklist* **is not empty**):

AST := *worklist*.remove();

if (AST **is complete &** AST **satisfies** E): **return** AST;

worklist.addAll(**expand**(AST));

Idea 2: BFS-Style Search

- Idea: simultaneously make progress on all branches
- Implementation
- Pros
  - Prioritize based on program size: find simplest solution first

Idea 2: BFS-Style Search

- Idea: simultaneously make progress on all branches
- Implementation
- Pros
  - Prioritize based on program size: find simplest solution first
  - Occam's Razor: "the simplest explanation is usually the best one"

Idea 2: BFS-Style Search

- Idea: simultaneously make progress on all branches
- Implementation
- Pros
  - Prioritize based on program size: find simplest solution first
  - Occam's Razor: "the simplest explanation is usually the best one"
- Potential problems?

Idea 2: BFS-Style Search

- Idea: simultaneously make progress on all branches
- Implementation
- Pros
  - Prioritize based on program size: find simplest solution first
  - Occam's Razor: "the simplest explanation is usually the best one"
- Potential problems?
  - Treat all operators more or less equally
  - But some operators may be less useful..

Search Prioritization

- Idea 1: DFS-style
- Idea 2: BFS-style
- Idea 3: Weighted search

Idea 3: Weighted Search

- Idea: different DSL operators have different weights

Idea 3: Weighted Search

- Idea: different DSL operators have different weights
  - Higher weights: higher costs
  - Cost of a program = “sum” of operator costs
  - Find a program with smallest cost

Idea 3: Weighted Search

- Idea: different DSL operators have different weights
  - Higher weights: higher costs
  - Cost of a program = "sum" of operator costs
  - Find a program with smallest cost
- Implementation

Top-Down-Search ($(T, N, P, S), E$):

worklist := { S }; // **Use a priority queue (smallest cost first)**

while (*worklist* **is not empty**):

AST := *worklist*.remove();

if (AST **is complete &** AST **satisfies** E): **return** AST;

worklist.addAll(**expand**(AST));

Idea 3: Weighted Search

- Implementation
  - Need to define cost for **partial** programs

Idea 3: Weighted Search

- Implementation
  - Need to define cost for **partial** programs
  - Use cost function (aka. ranking function, scoring function)

$Cost(s) = int$, where s is a grammar symbol

$Cost(f) = int$, where f is a DSL operator

$$Cost(f(P_1, \ldots, P_n)) = g\big(Cost(f), Cost(P_1), \ldots, Cost(P_n)\big)$$

Idea 3: Weighted Search

- Implementation
  - Need to define cost for **partial** programs
  - Use cost function (aka. ranking function, scoring function)

$Cost(s) = int$, where s is a grammar symbol

$Cost(f) = int$, where f is a DSL operator

$$Cost(f(P_1, \ldots, P_n)) = g\left(Cost(f), Cost(P_1), \ldots, Cost(P_n)\right)$$

  - Cost function definitions are very flexible, but they are typically “compositional”

Idea 3: Weighted Search

- Implementation
  - Need to define cost for **partial** programs
  - Use cost function (aka. ranking function, scoring function)

$Cost(s) = int$, where s is a grammar symbol

$Cost(f) = int$, where f is a DSL operator

$$Cost(f(P_1, \ldots, P_n)) = g\big(Cost(f), Cost(P_1), \ldots, Cost(P_n)\big)$$

  - Cost function definitions are very flexible, but they are typically “compositional”
  - Question: is it guaranteed to return a program **with smallest cost**?

Idea 3: Weighted Search

- Implementation
  - Question: is it guaranteed to return a program **with smallest cost**?

Top-Down-Search ($(T, N, P, S), E$):

worklist := { S }; // **Use a priority queue (smallest cost first)**

while (*worklist* **is not empty**):

AST := *worklist*.remove();

if (AST **is complete &** AST **satisfies** E): **return** AST;

worklist.addAll(**expand**(AST));

Idea 3: Weighted Search

- Implementation
  - Question: is it guaranteed to return a program **with smallest cost**?
  - Or: how to design cost function such that it's guaranteed?

Top-Down-Search ($(T, N, P, S), E$):

worklist := { S }; // **Use a priority queue (smallest cost first)**

while (*worklist* **is not empty**):

AST := *worklist*.remove();

if (AST **is complete &** AST **satisfies** E): **return** AST;

worklist.addAll(**expand**(AST));

Idea 3: Weighted Search

- Implementation
  - Question: is it guaranteed to return a program **with smallest cost**?
  - Or: how to design cost function such that it's guaranteed?
  - Here is one definition:

$$Cost(s) = 0 \qquad Cost(f) = 1 \qquad Cost(f(P_1, \ldots, P_n)) = Cost(f) + Cost(P_1) + \ldots + Cost(P_n)$$

Symbol s essentially corresponds to a "hole" in a partial program

Top-Down-Search ($(T, N, P, S), E$):

worklist := { S }; // **Use a priority queue (smallest cost first)**

while (*worklist* **is not empty**):

AST := *worklist*.remove();

if (AST **is complete &** AST **satisfies** E): **return** AST;

worklist.addAll(**expand**(AST));

Idea 3: Weighted Search

- Implementation
  - Question: is it guaranteed to return a program **with smallest cost**?
  - Or: how to design cost function such that it's guaranteed?
  - Here is one definition:

$$Cost(s) = 0 \qquad Cost(f) = 1 \qquad Cost(f(P_1, \ldots, P_n)) = Cost(f) + Cost(P_1) + \ldots + Cost(P_n)$$

  - Why?

Top-Down-Search ($(T, N, P, S), E$):

worklist := { S }; // **Use a priority queue (smallest cost first)**

while (*worklist* **is not empty**):

AST := *worklist*.remove();

if (AST **is complete &** AST **satisfies** E): **return** AST;

worklist.addAll(**expand**(AST));

Idea 3: Weighted Search

- Consider:

  $Cost(s) = 0 \qquad Cost(f) = 1$

  $Cost(f(P_1, \ldots, P_n)) = Cost(f) + Cost(P_1) + \ldots + Cost(P_n)$

- Why is smallest cost guaranteed?

Top-Down-Search ($(T, N, P, S), E$):

worklist := { S }; // **Use a priority queue (smallest cost first)**

while (*worklist* **is not empty**):

AST := *worklist*.remove();

if (AST **is complete &** AST **satisfies** E): **return** AST;

worklist.addAll(**expand**(AST));

Idea 3: Weighted Search

- Consider:

  $Cost(s) = 0$

  $Cost(f(P_1, \ldots, P_n)) = Cost(f) + Cost(P_1) + \ldots + Cost(P_n)$

- Why is smallest cost guaranteed?
  - Proof by contradiction

Top-Down-Search ($(T, N, P, S), E$):

worklist := { S }; // **Use a priority queue (smallest cost first)**

while (*worklist* **is not empty**):

AST := *worklist*.remove();

if (AST **is complete &** AST **satisfies** E): **return** AST;

worklist.addAll(**expand**(AST));

Idea 3: Weighted Search

- Consider:

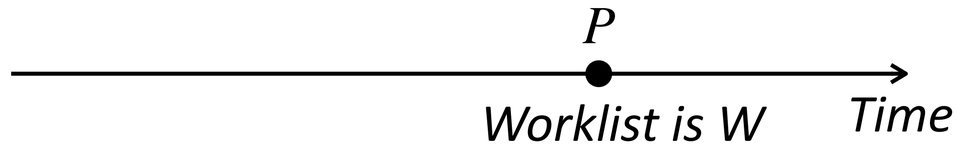
$$Cost(s) = 0$$

$$Cost(f(P_1, \ldots, P_n)) = Cost(f) + Cost(P_1) + \ldots + Cost(P_n)$$

- Why is smallest cost guaranteed?
  - Proof by contradiction
  - Suppose we synthesized program P using top-down search

Top-Down-Search ($(T, N, P, S), E$):

```
worklist := { S };  // Use a priority queue (smallest cost first)
while ( worklist is not empty ):
    AST := worklist.remove();
    if ( AST is complete & AST satisfies E ): return AST;
    worklist.addAll( expand( AST ) );
```



Idea 3: Weighted Search

- Consider:

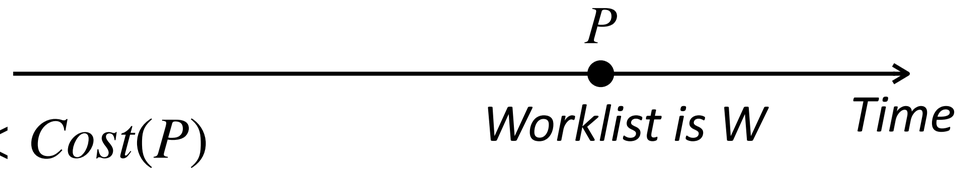
$$Cost(s) = 0$$

$$Cost(f(P_1, \ldots, P_n)) = Cost(f) + Cost(P_1) + \ldots + Cost(P_n)$$

- Why is smallest cost guaranteed?
  - Proof by contradiction
  - Suppose we synthesized program P using top-down search
  - Suppose there exists concrete program P' such that $Cost(P') < Cost(P)$

Top-Down-Search ($(T, N, P, S), E$):

```
worklist := { S };  // Use a priority queue (smallest cost first)
while ( worklist is not empty ):
    AST := worklist.remove();
    if ( AST is complete & AST satisfies E ): return AST;
    worklist.addAll( expand( AST ) );
```



Idea 3: Weighted Search

- Consider:

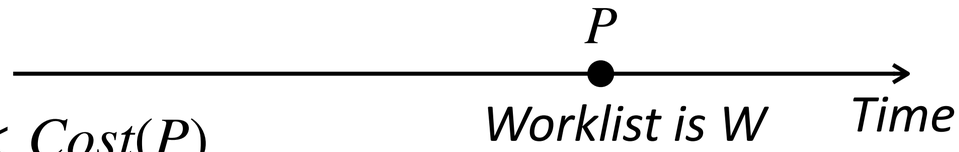
$$Cost(s) = 0$$

$$Cost(f(P_1, \ldots, P_n)) = Cost(f) + Cost(P_1) + \ldots + Cost(P_n)$$

- Why is smallest cost guaranteed?
  - Proof by contradiction
  - Suppose we synthesized program P using top-down search
  - Suppose there exists concrete program P' such that $Cost(P') < Cost(P)$
  - Claim: P' must not be in *W*

Top-Down-Search ($(T, N, P, S), E$):

```
worklist := { S };  // Use a priority queue (smallest cost first)
while ( worklist is not empty ):
    AST := worklist.remove();
    if ( AST is complete & AST satisfies E ): return AST;
    worklist.addAll( expand( AST ) );
```



Idea 3: Weighted Search

- Consider:

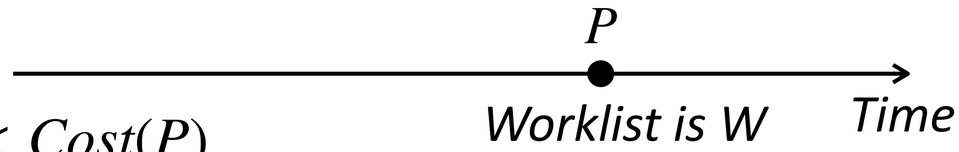
$$Cost(s) = 0$$

$$Cost(f(P_1, \ldots, P_n)) = Cost(f) + Cost(P_1) + \ldots + Cost(P_n)$$

- Why is smallest cost guaranteed?
  - Proof by contradiction
  - Suppose we synthesized program P using top-down search
  - Suppose there exists concrete program P' such that $Cost(P') < Cost(P)$
  - Claim: P' must not be in *W*
  - Find partial program P'' in *W* from which P' can be derived

Top-Down-Search ($(T, N, P, S), E$):

```
worklist := { S };  // Use a priority queue (smallest cost first)
while ( worklist is not empty ):
    AST := worklist.remove();
    if ( AST is complete & AST satisfies E ): return AST;
    worklist.addAll( expand( AST ) );
```



Idea 3: Weighted Search

- Consider:

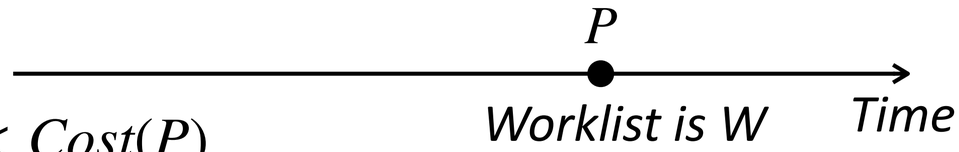
  $Cost(s) = 0$

  $Cost(f(P_1, \ldots, P_n)) = Cost(f) + Cost(P_1) + \ldots + Cost(P_n)$

- Why is smallest cost guaranteed?
  - Proof by contradiction
  - Suppose we synthesized program P using top-down search
  - Suppose there exists concrete program P' such that $Cost(P') < Cost(P)$
  - Claim: P' must not be in *W*
  - Find partial program P'' in *W* from which P' can be derived
  - Claim: $Cost(P'') \leq Cost(P')$

Top-Down-Search ($(T, N, P, S), E$):

```
worklist := { S };  // Use a priority queue (smallest cost first)
while ( worklist is not empty ):
    AST := worklist.remove();
    if ( AST is complete & AST satisfies E ): return AST;
    worklist.addAll( expand( AST ) );
```



Idea 3: Weighted Search

- Consider:

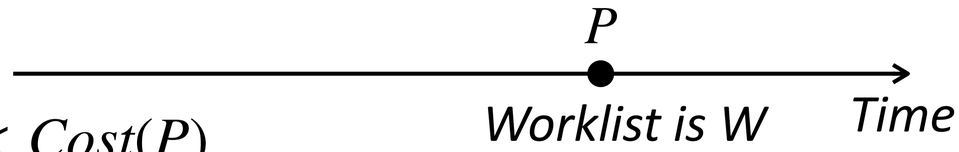
  $Cost(s) = 0$

  $Cost(f(P_1, \ldots, P_n)) = Cost(f) + Cost(P_1) + \ldots + Cost(P_n)$

- Why is smallest cost guaranteed?
  - Proof by contradiction
  - Suppose we synthesized program P using top-down search
  - Suppose there exists concrete program P' such that $Cost(P') < Cost(P)$
  - Claim: P' must not be in *W*
  - Find partial program P'' in *W* from which P' can be derived
  - Claim: $Cost(P'') \leq Cost(P')$
  - Claim: $Cost(P) \leq Cost(P'')$

Top-Down-Search ($(T, N, P, S), E$):

```
worklist := { S };  // Use a priority queue (smallest cost first)
while ( worklist is not empty ):
    AST := worklist.remove();
    if ( AST is complete & AST satisfies E ): return AST;
    worklist.addAll( expand( AST ) );
```



Idea 3: Weighted Search

- Consider:

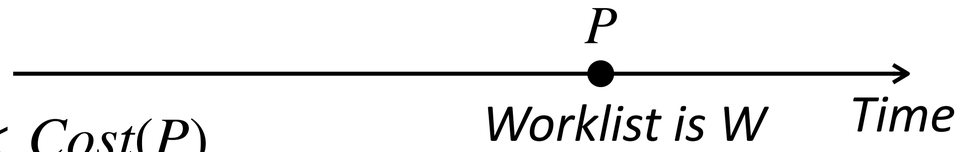
  $Cost(s) = 0$

  $Cost(f(P_1, \ldots, P_n)) = Cost(f) + Cost(P_1) + \ldots + Cost(P_n)$

- Why is smallest cost guaranteed?
  - Proof by contradiction
  - Suppose we synthesized program P using top-down search
  - Suppose there exists concrete program P' such that $Cost(P') < Cost(P)$
  - Claim: P' must not be in *W*
  - Find partial program P'' in *W* from which P' can be derived
  - Claim: $Cost(P'') \leq Cost(P')$
  - Claim: $Cost(P) \leq Cost(P'')$
  - Claim: $Cost(P) \leq Cost(P')$

Top-Down-Search ($(T, N, P, S), E$):

```
worklist := { S };  // Use a priority queue (smallest cost first)
while ( worklist is not empty ):
    AST := worklist.remove();
    if ( AST is complete & AST satisfies E ): return AST;
    worklist.addAll( expand( AST ) );
```



Idea 3: Weighted Search

- Consider:

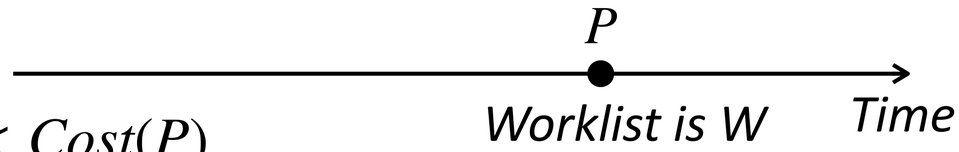
  $Cost(s) = 0$

  $Cost(f(P_1, \ldots, P_n)) = Cost(f) + Cost(P_1) + \ldots + Cost(P_n)$

- Why is smallest cost guaranteed?
  - Proof by contradiction
  - Suppose we synthesized program P using top-down search
  - Suppose there exists concrete program P' such that $Cost(P') < Cost(P)$
  - Claim: P' must not be in *W*
  - Find partial program P'' in *W* from which P' can be derived
  - Claim: $Cost(P'') \leq Cost(P')$
  - Claim: $Cost(P) \leq Cost(P'')$
  - Claim: $Cost(P) \leq Cost(P')$
  - **Contradiction!**

Top-Down-Search ($(T, N, P, S), E$):

```
worklist := { S };  // Use a priority queue (smallest cost first)
while ( worklist is not empty ):
    AST := worklist.remove();
    if ( AST is complete & AST satisfies E ): return AST;
    worklist.addAll( expand( AST ) );
```

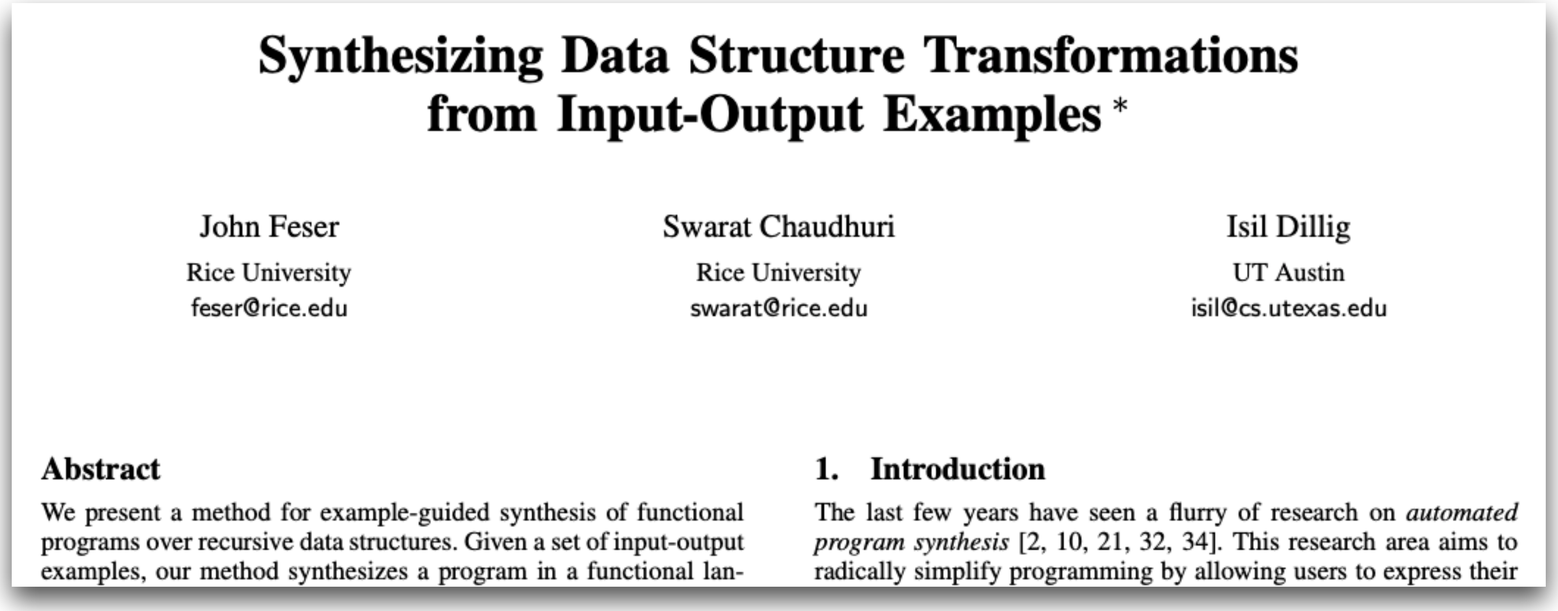


Idea 3: Weighted Search

- Idea: different DSL operators have different weights
  - Use cost function (aka. ranking function, scoring function)
- Implementation
- Pros
  - more general than BFS/DFS
  - Used by many synthesizers

Idea 3: Weighted Search

- Pros
  - more general than BFS/DFS
  - Used by many synthesizers



Synthesizing Data Structure Transformations from Input-Output Examples *

John Feser
Rice University
feser@rice.edu

Swarat Chaudhuri
Rice University
swarat@rice.edu

Isil Dillig
UT Austin
isil@cs.utexas.edu

Abstract

We present a method for example-guided synthesis of functional programs over recursive data structures. Given a set of input-output examples, our method synthesizes a program in a functional lan-

1. Introduction

The last few years have seen a flurry of research on *automated program synthesis* [2, 10, 21, 32, 34]. This research area aims to radically simplify programming by allowing users to express their

Idea 3: Weighted Search

- Pros
  - more general than BFS/DFS
  - Used by many synthesizers

Cost model Each program e in the language has a *cost* $\mathcal{C}(e) \geq 0$. This cost is defined inductively. Specifically, we assume that each primitive operator $\oplus$ and constant c has a known, positive cost. Costs for more complex expressions satisfy constraints like the following (we skip some of the cases for brevity):

- $\mathcal{C}(\oplus\ e) > \mathcal{C}(\oplus) + \mathcal{C}(e)$
- $\mathcal{C}(\lambda x.e) > \mathcal{C}(e)$
- $\mathcal{C}(e_1\ e_2) > \mathcal{C}(e_1) + \mathcal{C}(e_2)$
- $\mathcal{C}(x) = 0$. Intuitively, we assign costs to the *definition*, rather than the *use*, of variables.

Synthesizing Data Structure Transf
from Input-Output Exampl

John Feser
Rice University
feser@rice.edu

Swarat Chaudhuri
Rice University
swarat@rice.edu

Isil Dillig
UT Austin
isil@cs.utexas.edu

Abstract

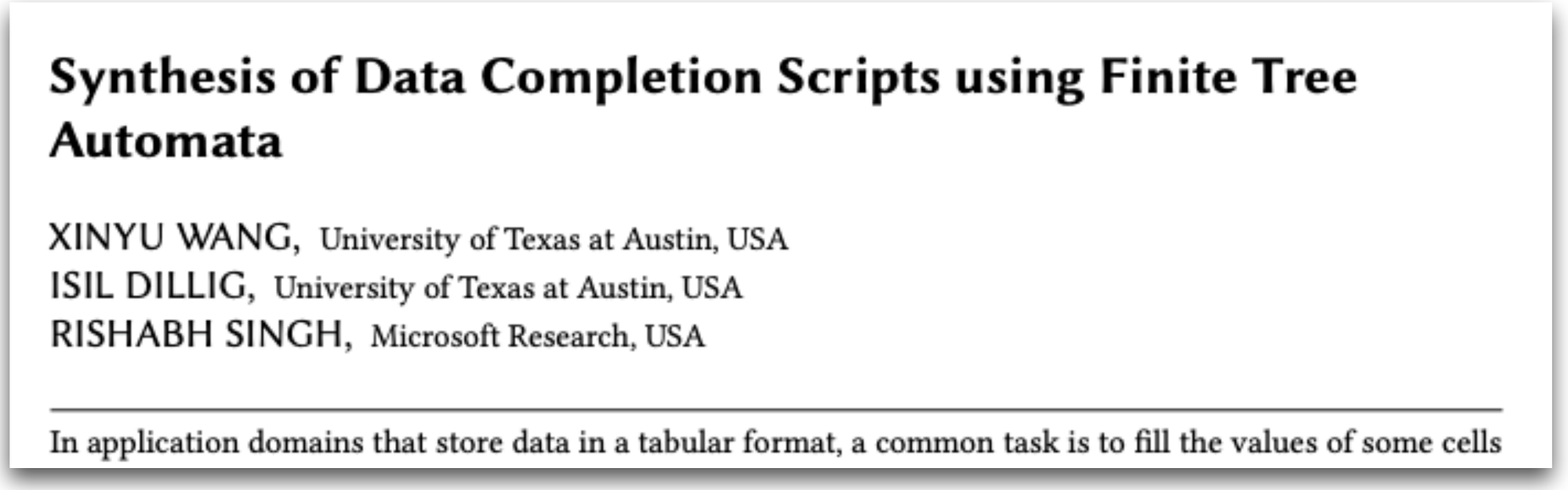
We present a method for example-guided synthesis of functional programs over recursive data structures. Given a set of input-output examples, our method synthesizes a program in a functional lan-

1. Introduction

The last few years have seen a flurry of research on *automated program synthesis* [2, 10, 21, 32, 34]. This research area aims to radically simplify programming by allowing users to express their

Idea 3: Weighted Search

- Pros
  - more general than BFS/DFS
  - Used by many synthesizers



Synthesis of Data Completion Scripts using Finite Tree Automata

XINYU WANG, University of Texas at Austin, USA
ISIL DILLIG, University of Texas at Austin, USA
RISHABH SINGH, Microsoft Research, USA

In application domains that store data in a tabular format, a common task is to fill the values of some cells

Idea 3: W

- Pros
  - more general than BFS/DFS
  - Used by many synthesizers

Synthesis of Data Completion Scripts Automata

XINYU WANG, University of Texas at Austin, USA
ISIL DILLIG, University of Texas at Austin, USA
RISHABH SINGH, Microsoft Research, USA

In application domains that store data in a tabular format, a comm

A HEURISTIC SCORING FUNCTION

Recall that our synthesis algorithm uses a scoring function θ to choose between multiple programs that satisfy the input-output examples. The design of the scoring fuction follows the *Occam's razor* principle and tries to favor simpler, more general programs over complex ones.

In more detail, our scoring function assign scores to constants, cell mappers, and predicates in our DSL in a way that satisfies the following properties:

- A predicate with mapper $\lambda c.c$ has a higher score than the same predicate with other mappers.
- For predicates containing the same mapper χ, we require that the scoring function satisfies the following constraint:

$$\theta(\text{True}) > \theta(\text{Val}(\chi(z)) = ?) > \theta(\text{Val}(\chi(y)) = \text{Val}(\chi(z))) > \theta(\text{Val}(\chi(z)) = s) = \theta(\text{Val}(\chi(z)) \neq s)$$

- $\theta(\phi_1, \cdot\cdot, \phi_n)$ takes into account both the scores of each conjunct as well as the number of conjuncts. That is, θ assigns a higher score to predicates that have conjuncts with higher scores, and assigns lower scores to predicates with more conjuncts. One design choice satisfying this criterion is to take the average of scores of all the terms in the conjunct.
- For scores of integer k we have $\theta(1) > \theta(2) > \theta(-1) > \theta(3) > \theta(-2) > \theta(-3)$.

Using the scores assigned to predicates, mappers, and constants, we then assign scores to more complex programs in the DSL in the following way. The score of a GetCell program is defined by taking into account both the scores of its arguments and the recursion depth (number of nested GetCell constructs). One possible way to assign scores to GetCell programs is therefore the following:

$$\theta(\text{GetCell}(x, \text{dir}, k, \lambda y \lambda z.\phi)) := \theta(k) \cdot \theta(\phi)$$

$$\theta(\text{GetCell}(\tau, \text{dir}, k, \lambda y \lambda z.\phi)) := \frac{\theta(\tau) + \theta(k) \cdot \theta(\phi)}{depth(\tau)}$$

$$depth(x) := 0$$

$$depth(\text{GetCell}(\tau, \text{dir}, k, \lambda y \lambda z.\phi)) := depth(\tau) + 1$$

The score of a simple program, i.e., $\text{List}(\tau_1, \cdot\cdot, \tau_n)$ or $\text{Filter}(\tau_1, \tau_2, \tau_3, \lambda y \lambda z.\phi)$, takes into account of the scores of its arguments and the number of the arguments. Specifically, it assigns scores in the following way:

$$\theta(\text{List}(\tau_1, \cdot\cdot, \tau_n)) := \frac{\theta(\tau_1) + \cdot\cdot + \theta(\tau_n)}{n}$$

$$\theta(\text{Filter}(\tau_1, \tau_2, \tau_3, \lambda y \lambda z.\phi)) := \frac{\theta(\tau_1) + \theta(\tau_2) + \theta(\tau_3)}{3}$$

Idea 3: Weighted Search

- Idea: different DSL operators have different weights
  - Use cost function (aka. ranking function, scoring function)
- Implementation
- Pros
  - more general than BFS/DFS
  - Used by many synthesizers
- Cons?

Idea 3: Weighted Search

- Idea: different DSL operators have different weights
  - Use cost function (aka. ranking function, scoring function)
- Implementation
- Pros
  - more general than BFS/DFS
  - Used by many synthesizers
- Cons?
  - “cost” $\approx$ “program size”
  - So, still find program with smallest “size”
  - Essentially still use “size” as proxy for “likelihood of satisfying spec”

Search Prioritization

- Idea 1: DFS-style
- Idea 2: BFS-style
- Idea 3: Weighted search
- Idea 4: Use statistical models

Idea 4: Use Statistical Models

- Idea: explore programs in the order of **likelihood**, instead of size

Idea 4: Use Statistical Models

- Idea: explore programs in the order of **likelihood**, instead of size
  - Prioritize branches that are more likely to lead to correct programs

Idea 4: Use Statistical Models

- Idea: explore programs in the order of **likelihood**, instead of size
  - Prioritize branches that are more likely to lead to correct programs
- Learn a statistical (probabilistic) model from a corpus of programs
  - Likelihood of a branch is measured quantitatively by the model

Idea 4: Use Statistical Models

- Idea: explore programs in the order of **likelihood**, instead of size
  - Prioritize branches that are more likely to lead to correct programs
- Learn a statistical (probabilistic) model from a corpus of programs
  - Likelihood of a branch is measured quantitatively by the model
- Implementation

Top-Down-Search ($(T, N, P, S), E$):

worklist := { S }; // **Priority queue but costs are given by model**

while (*worklist* **is not empty**):

AST := *worklist*.remove();

if (AST **is complete &** AST **satisfies** E): **return** AST;

worklist.addAll(**expand**(AST));

Idea 4: Use Statistical Models

- Idea: explore programs in the order of **likelihood**, instead of size
- Learn a statistical (probabilistic) model from a corpus of programs
- Implementation
  - Key: learn a statistical **model** from a **corpus** of programs

Idea 4: Use Statistical Models

- Idea: explore programs in the order of **likelihood**, instead of size
- Learn a statistical (probabilistic) model from a corpus of programs
- Implementation
  - Key: learn a statistical **model** from a **corpus** of programs
  - Data:
  - Models:

Idea 4: Use Statistical Models

- Idea: explore programs in the order of **likelihood**, instead of size
- Learn a statistical (probabilistic) model from a corpus of programs
- Implementation
  - Key: learn a statistical **model** from a **corpus** of programs
  - Data:
    - Programs written by real-world users (e.g., from Github repos, MOOCs)
    - Synthetic data (e.g., programs from DSL)
  - Models:

Idea 4: Use Statistical Models

- Idea: explore programs in the order of **likelihood**, instead of size
- Learn a statistical (probabilistic) model from a corpus of programs
- Implementation
  - Key: learn a statistical **model** from a **corpus** of programs
  - Data:
    - Programs written by real-world users (e.g., from Github repos, MOOCs)
    - Synthetic data (e.g., programs from DSL)
  - Models:
    - N-gram models
    - Neural networks

Use N-gram Models

- Statistical Language Models
  - Probability distribution over sentences in a language: $Prob(s)$ for $s \in L$
  - Used in Natural Language Processing

Use N-gram Models

- Statistical Language Models
  - Probability distribution over sentences in a language: $Prob(s)$ for $s \in L$
  - Used in Natural Language Processing
- Concept in computational linguistics and probability
- An n-gram is a sequence of n items from a given sample of text or speech
  - 1 item: unigram
  - 2 items: bigram
  - 3 items: trigram
- An n-gram model predicts the next item given the first n-1 items
- One can train such models from data

Use N-gram Models

- The quick brown ? jumped.
  - ? = Fox
  - ? = Dog
  - ? = Computer

Use N-gram Models

- The quick brown ? jumped
  - ? = Fox
  - ? = Dog
  - ? = Computer

- 2-gram model predicts the next word based on the current word

Use N-gram Models

- The quick brown ? jumped
  - ? = Fox
  - ? = Dog
  - ? = Computer

- 2-gram model predicts the next word based on the current word
  - Model is trained from pairs of words (training data): <brown, fox>, <brown, dog>, ..
  - If word preceding ? is a, fill the hole with word x, such that <a, x> is bigram in the training data

Use N-gram Models

- Programs = Sentences
- Predict parts of program given other parts

Use N-gram Models

- Programs = Sentences
- Predict parts of program given other parts
- Idea first explored in SLANG for code completion (which is a form of program synthesis)

Code Completion with Statistical Language Models

Veselin Raychev
ETH Zürich
veselin.raychev@inf.ethz.ch

Martin Vechev
ETH Zürich
martin.vechev@inf.ethz.ch

Eran Yahav
Technion
yahave@cs.technion.ac.il

Abstract

We address the problem of synthesizing code completions for programs using APIs. Given a program with holes, we synthesize completions for holes with the most likely sequences of method calls.

Our main idea is to reduce the problem of code completion to a natural-language processing problem of predicting probabilities of sentences. We design a simple and scalable static analysis that extracts sequences of method calls from a large codebase, and

Our Approach: Synthesis with Statistical Language Models Statistical language models have been successfully used to model regularities in natural languages and applied to problems such as speech recognition, optical character recognition, and others [28].

Our main idea is to reduce the problem of code completion to a *natural-language processing* problem of predicting probabilities of sentences: we use regularities found in sequences of method invocations to predict and synthesize a likely method invocation

SLANG

```
SmsManager smsMgr = SmsManager.getDefault();
int length = message.length();
if (length > MAX_SMS_MESSAGE_LENGTH) {
  ArrayList<String> msgList =
      smsMgr.divideMsg(message);
  ? {smsMgr, msgList}  // (H1)
} else {
  ? {smsMgr, message}  // (H2)
}
```

```
SmsManager smsMgr = SmsManager.getDefault();
int length = message.length();
if (length > MAX_SMS_MESSAGE_LENGTH) {
  ArrayList<String> msgList =
      smsMgr.divideMsg(message);
  smsMgr.sendMultipartTextMessage(...msgList...);
} else {
  smsMgr.sendTextMessage(...message...);
}
```

Input: incomplete code snippet

Automatically find completions using statistical model

Output: completed code

Use N-gram Models

- Idea first explored in SLANG for code completion (which is a form of program synthesis)
- Idea then applied in many program synthesizers such as Morpheus

Component-Based Synthesis of Table Consolidation and Transformation Tasks from Examples *

Yu Feng
University of Texas at Austin, USA
yufeng@cs.utexas.edu

Ruben Martins
University of Texas at Austin, USA
rmartins@cs.utexas.edu

Jacob Van Geffen
University of Texas at Austin, USA
jsv@cs.utexas.edu

Isil Dillig
University of Texas at Austin, USA
isil@cs.utexas.edu

Swarat Chaudhuri
Rice University, USA
swarat@rice.edu

Abstract

This paper presents a novel component-based synthesis algorithm that marries the power of type-directed search

Keywords Program synthesis, Programming by example, data preparation, Component-based synthesis, SMT-based deduction

Morpheus

- Synthesize R programs from input-output examples
  - Top-down search
  - SMT-based deduction for pruning
  - **2-gram model for prioritization**

Morpheus

- Synthesize R programs from input-output examples
  - Top-down search
  - SMT-based deduction for pruning
  - **2-gram model for prioritization**

Recall from Section 5 that MORPHEUS uses a cost model for picking the "best" hypothesis from the worklist. Inspired by previous work on code completion [28], we use a cost model based on a statistical analysis of existing code. Specifically, MORPHEUS analyzes existing code snippets that use components from Λ_T and represents each snippet as a 'sentence' where 'words' correspond to components in Λ_T. Given this representation, MORPHEUS uses the 2-gram model in SRILM [34] to assign a score to each hypothesis. Specifically, we train our language model by collecting approximately 15,000 code snippets from Stackoverflow using the search keywords `tidyr` and `dplyr`. For each code snippet, we ignore its control flow and represent it using a "sentence" where each "word" corresponds to an API call. Based on this training data, the hypotheses in the worklist W from Algorithm 1 are then ordered using the scores obtained from the n-gram model.

Morpheus

- Synthesize R programs from input-output examples
  - Top-down search
  - SMT-based deduction for pruning
  - **2-gram model for prioritization**
    - Data: StackOverflow

Recall from Section 5 that MORPHEUS uses a cost model for picking the "best" hypothesis from the worklist. Inspired by previous work on code completion [28], we use a cost model based on a statistical analysis of existing code. Specifically, MORPHEUS analyzes existing code snippets that use components from Λ_T and represents each snippet as a 'sentence' where 'words' correspond to components in Λ_T. Given this representation, MORPHEUS uses the 2-gram model in SRILM [34] to assign a score to each hypothesis. Specifically, we train our language model by collecting approximately 15,000 code snippets from Stackoverflow using the search keywords `tidyr` and `dplyr`. For each code snippet, we ignore its control flow and represent it using a "sentence" where each "word" corresponds to an API call. Based on this training data, the hypotheses in the worklist W from Algorithm 1 are then ordered using the scores obtained from the n-gram model.

Morpheus

- Synthesize R programs from input-output examples
  - Top-down search
  - SMT-based deduction for pruning
  - **2-gram model for prioritization**
    - Data: StackOverflow
    - Partial programs are ordered according to the model

Recall from Section 5 that MORPHEUS uses a cost model for picking the "best" hypothesis from the worklist. Inspired by previous work on code completion [28], we use a cost model based on a statistical analysis of existing code. Specifically, MORPHEUS analyzes existing code snippets that use components from Λ_T and represents each snippet as a 'sentence' where 'words' correspond to components in Λ_T. Given this representation, MORPHEUS uses the 2-gram model in SRILM [34] to assign a score to each hypothesis. Specifically, we train our language model by collecting approximately 15,000 code snippets from Stackoverflow using the search keywords `tidyr` and `dplyr`. For each code snippet, we ignore its control flow and represent it using a "sentence" where each "word" corresponds to an API call. Based on this training data, the hypotheses in the worklist W from Algorithm 1 are then ordered using the scores obtained from the n-gram model.

Morpheus

- Synthesize R programs from input-output examples
  - Top-down search
  - SMT-based deduction for pruning
  - **2-gram model for prioritization**
    - Works well in practice

Recall from Section 5 that MORPHEUS uses a cost model for picking the "best" hypothesis from the worklist. Inspired by previous work on code completion [28], we use a cost model based on a statistical analysis of existing code. S sting code snippets t resents each snippet a nd to components in M EUS uses the 2-gram n re to each hypothe- s model by collecting a m Stackoverflow us- i lyr. For each code s represent it using a " onds to an API call. Based on this training data, the hypotheses in the worklist W from Algorithm 1 are then ordered using the scores obtained from the n-gram model.

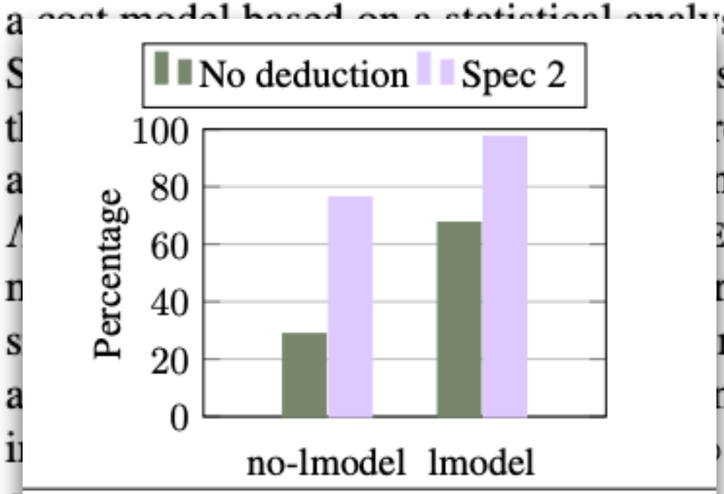


Figure 18. Impact of language model

Morpheus

- Synthesize R programs from input-output examples
  - Top-down search
  - SMT-based deduction for pruning
  - **2-gram model for prioritization**
    - Works well in practice
- Discuss Morpheus next lecture

Recall from Section 5 that MORPHEUS uses a cost model for picking the "best" hypothesis from the worklist. Inspired by previous work on code completion [28], we use a cost model based on a statistical analysis of existing code. S sting code snippets t resents each snippet a nd to components in M US uses the 2-gram n re to each hypothe- s model by collecting a m Stackoverflow us- i lyr. For each code s represent it using a " onds to an API call. Based on this training data, the hypotheses in the worklist W from Algorithm 1 are then ordered using the scores obtained from the n-gram model.

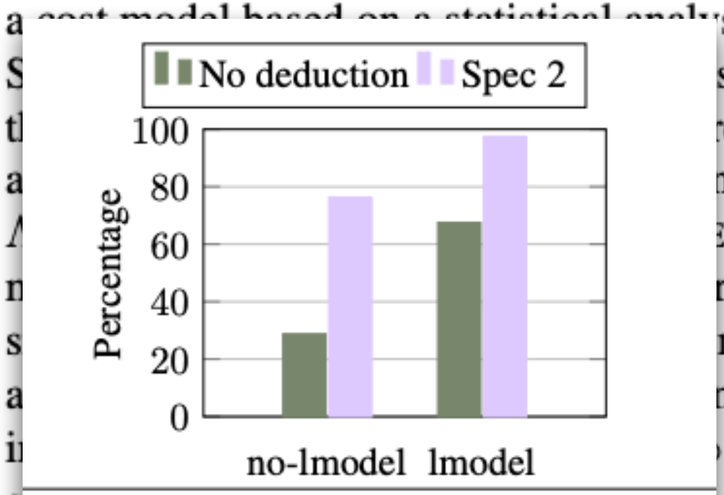


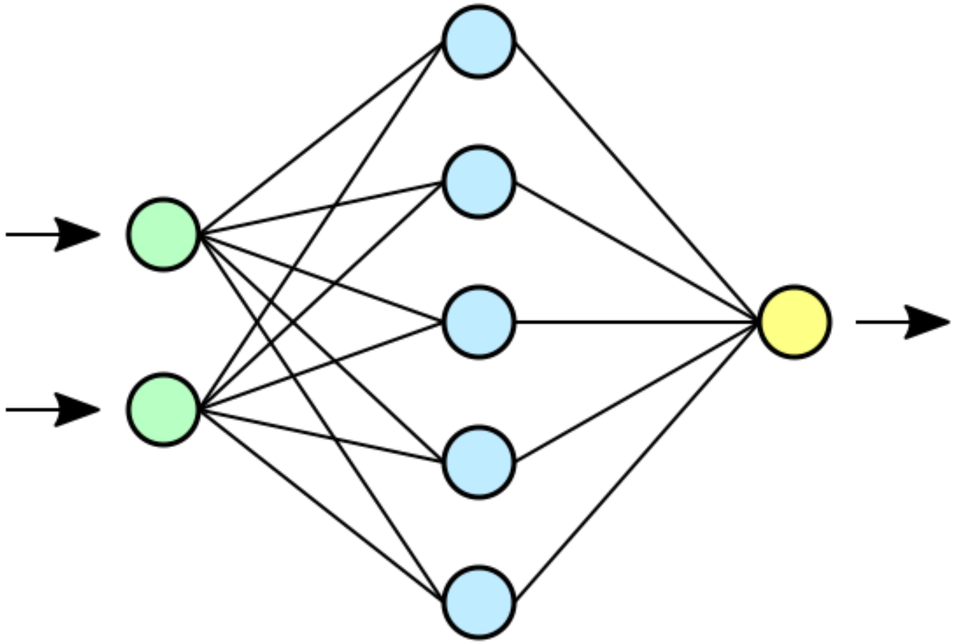
Figure 18. Impact of language model

Idea 4: Use Statistical Models

- Idea 1: DFS-style
- Idea 2: BFS-style
- Idea 3: Weighted search
- Idea 4: Statistical models
  - N-gram models
  - Neural networks

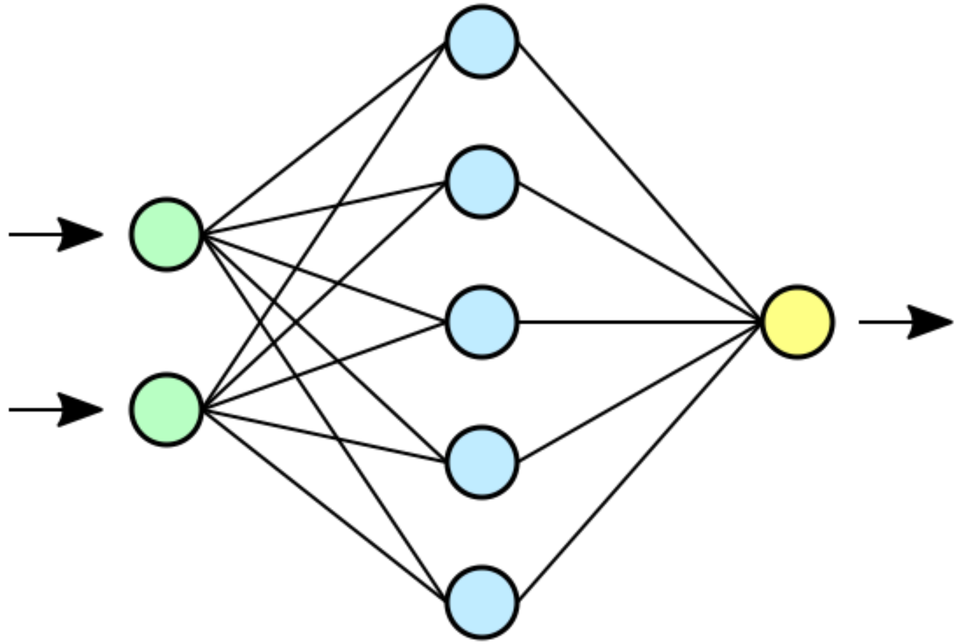
Use Neural Nets

- Neural net: neurons connected to each other that collectively perform certain tasks
  - Image recognition
  - Language translation
  - Medical diagnosis
  - Etc.



Use Neural Nets

- Neural net: neurons connected to each other that collectively perform certain tasks
  - Image recognition
  - Language translation
  - Medical diagnosis
  - Etc.
- First train a net, then use it for predictions



Use Neural Nets

- Idea first explored in DeepCoder

DEEPCODER: LEARNING TO WRITE PROGRAMS

Matej Balog*
Department of Engineering
University of Cambridge

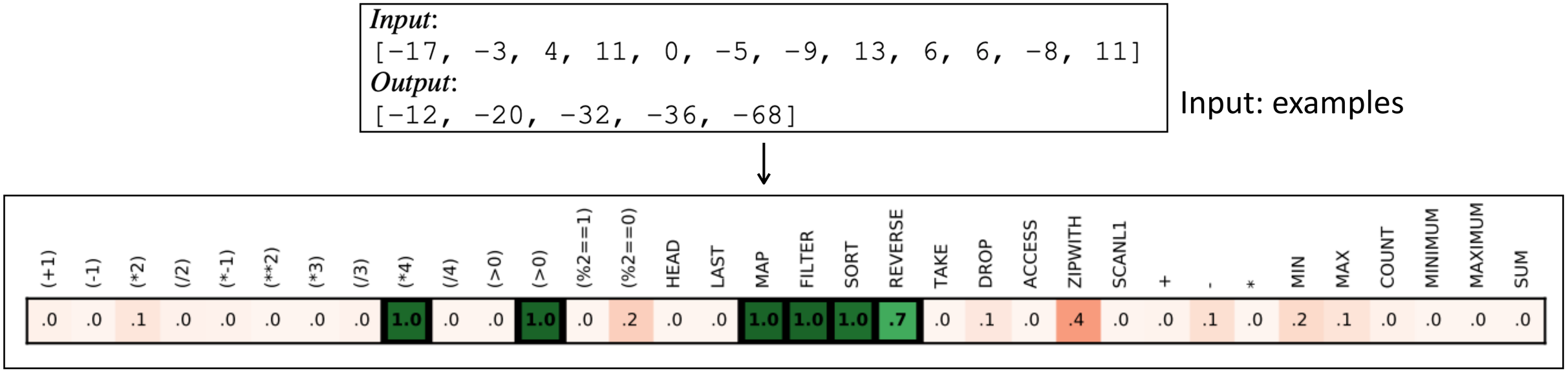
Alexander L. Gaunt, Marc Brockschmidt,
Sebastian Nowozin, Daniel Tarlow
Microsoft Research

ABSTRACT

We develop a first line of attack for solving programming competition-style problems from input-output examples using deep learning. The approach is to train a neural network to predict properties of the program that generated the outputs from the inputs. We use the neural network's predictions to augment search techniques from the programming languages community, including enumerative search and an SMT-based solver. Empirically, we show that our approach leads to an order of magnitude speedup over the strong non-augmented baselines and a Recurrent Neural Network approach, and that we are able to solve problems of difficulty comparable to the simplest problems on programming competition websites.

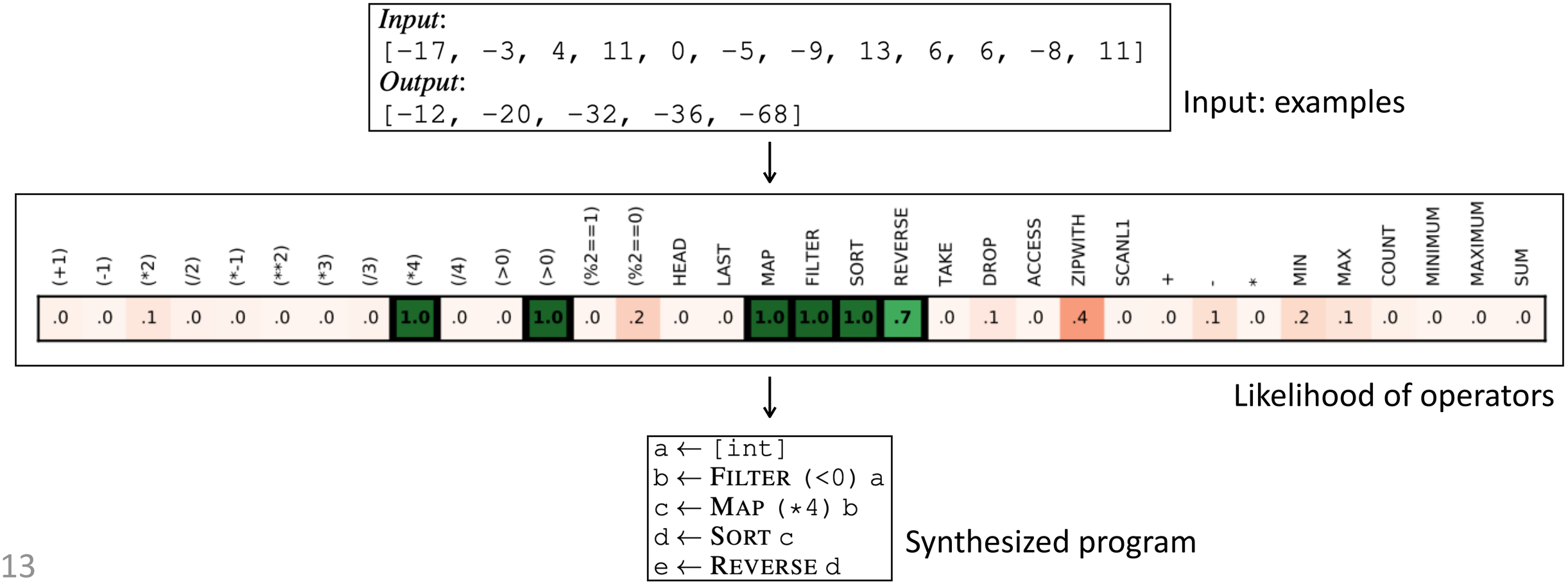
Use Neural Nets

- Idea first explored in DeepCoder
  - First, given examples, neural net predicts “properties” of desired programs



Use Neural Nets

- Idea first explored in DeepCoder
  - First, given examples, neural net predicts “properties” of desired programs
  - Then, use this information to search



Use Neural Nets

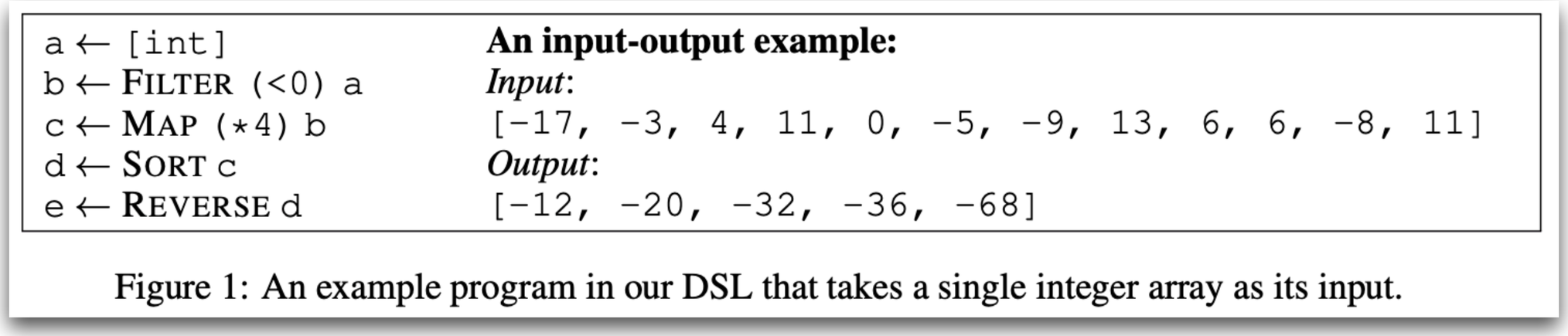
- Idea first explored in DeepCoder
  - Predict a distribution $q(a \mid E)$ of properties given examples E, then search over programs P ordered by $q(A(P) \mid E)$.

Use Neural Nets

- Idea first explored in DeepCoder
  - Predict a distribution $q(a \mid E)$ of properties given examples E, then search over programs P ordered by $q(A(P) \mid E)$.
  - Training: Generate (P, a, E) tuples where P is a program, a is a set of properties, E is a set of examples
    - Synthetic data: first construct programs, then construct examples for programs

Use Neural Nets

- Idea first explored in DeepCoder
  - Predict a distribution $q(a \mid E)$ of properties given examples E, then search over programs P ordered by $q(A(P) \mid E)$.
  - Training: Generate (P, a, E) tuples where P is a program, a is a set of properties, E is a set of examples
    - Synthetic data: first construct programs, then construct examples for programs



```
a ← [int]
b ← FILTER (<0) a
c ← MAP (*4) b
d ← SORT c
e ← REVERSE d
```

An input-output example:
Input:
`[-17, -3, 4, 11, 0, -5, -9, 13, 6, 6, -8, 11]`
Output:
`[-12, -20, -32, -36, -68]`

Figure 1: An example program in our DSL that takes a single integer array as its input.

Idea 4: Use Statistical Models

- Idea 1: DFS-style
- Idea 2: BFS-style
- Idea 3: Weighted search
- Idea 4: Statistical models
  - N-gram models
  - Neural networks
  - Others (see resources from course schedule)

Summary

- Search prioritization
  - DFS-style: have to use bound
  - BFS-style: Occam's razor
  - Cost function (ranking, scoring): more general
  - Statistical models: likelihood of satisfying spec