Path-based Inductive Synthesis for Program Inversion

Saurabh Srivastava [a]
Sumit Gulwani [b]
Swarat Chaudhuri [c]
Jeffrey S. Foster [d]

[a] University of California, Berkeley
[b] Microsoft Research, Redmond
[c] Rice University
[d] University of Maryland, College Park

Program Inversion as Synthesis

- **Task**

  Given a program ***P***, synthesis $\boldsymbol{P^{-1}}$ such that $\boldsymbol{P^{-1}}(\boldsymbol{P}(x)) = x$

- **Motivation**: Many common program/inverse pairs
  - Compress/decompress, insert/delete, lossless encode/decode, encrypt/decrypt, rollback, many more
  - Only having to write one increases productivity, reduces bugs

- **Problem**
  - Existing synthesis techniques not well-suited for inversion
  - Dedicated inversion techniques limited in scope

PINS: Path-based Inductive Synthesis

- **Specification**
  - Program to be inverted
  - Template hints: Control flow, and expressions, predicates
  - Functional requirement: Program + Inverse = Identity
- **Engine**: SMT solver (Z3)
- **Algorithm**: Inspired by testing
  - Explore path through program + template
  - Ask engine for instantiations on path to match spec
  - Iterate, refining space

Small path-bound hypothesis

"Program behavior can be summarized by examining a **carefully chosen**, **small, finite** set of paths"

- Same hypothesis underlies program testing
- As in testing, two questions:
  1) Which paths?
     - Especially since the template describes "set of programs"
  2) How can we ensure the generated inverse is correct?
     - We check using: manual inspection, testing, bounded verification

Example of templates

In-place run-length encoding:

A = [1,1,1,0,0,2,2,2,2]

↓

A = [1,0,2]
N = [3,2,4]

↓

A′ = [1,1,1,0,0,2,2,2,2]

Example of templates

In-place run-length encoding:

A = [1,1,1,0,0,2,2,2,2]

↓ Original encoder

A = [1,0,2]
N = [3,2,4]

↓

A′ = [1,1,1,0,0,2,2,2,2]

```
assume(n>=0);
i, m := 0, 0;        // parallel assignment
while (i < n)
    r := 1;
    while (i+1 < n && A[i] = A[i+1])
        r, i := r + 1, i+1;
    A[m], N[m], m, i := A[i], r, m+1, i+1;
```

Example of templates

In-place run-length encoding:

A = [1,1,1,0,0,2,2,2,2]

Original encoder

```
assume(n>=0);
i, m := 0, 0;        // parallel assignment
while (i < n)
  r := 1;
  while (i+1 < n && A[i] = A[i+1])
    r, i := r + 1, i+1;
  A[m], N[m], m, i := A[i], r, m+1, i+1;
```

A = [1,0,2]
N = [3,2,4]

Template decoder

```
i', m' := e_1, e_2;   // e_i ∈ E
while (p_1)           // p_i ∈ P
  r' := e_3;
  while (p_2)
    r', i', A' := e_4, e_5, e_6;
  m' := e_7;
```

A′ = [1,1,1,0,0,2,2,2,2]

Example of templates

In-place run-length encoding:

A = [1,1,1,0,0,2,2,2,2]

↓ Original encoder

A = [1,0,2]
N = [3,2,4]

↓ Template decoder

A′ = [1,1,1,0,0,2,2,2,2]

```
assume(n>=0);
i, m := 0, 0;        // parallel assignment
while (i < n)
   r := 1;
   while (i+1 < n && A[i] = A[i+1])
      r, i := r + 1, i+1;
   A[m], N[m], m, i := A[i], r, m+1, i+1;
```

```
i', m' := e1, e2;    // ei ∈ E
while (p1)           // pi ∈ P
   r' := e3;
   while (p2)
      r', i', A' := e4, e5, e6;
   m' := e7;
```

E = {
0, 1, m'+1, m'-1, r'+1, r'-1,
i'+1, i'-1, A'[m']:=A[i'],
A'[i'] := A[m'], N[m']
}

Example of templates

In-place run-length encoding:

A = [1,1,1,0,0,2,2,2,2]

Original encoder

A = [1,0,2]
N = [3,2,4]

Template decoder

A′ = [1,1,1,0,0,2,2,2,2]

```
assume(n>=0);
i, m := 0, 0;       // parallel assignment
while (i < n)
  r := 1;
  while (i+1 < n && A[i] = A[i+1])
    r, i := r + 1, i+1;
  A[m], N[m], m, i := A[i], r, m+1, i+1;
```

```
i', m' := e1, e2;   // ei ∈ E
while (p1)          // pi ∈ P
  r' := e3;
  while (p2)
    r', i', A' := e4, e5, e6;
  m' := e7;
```

E = {
0, 1, m'+1, m'-1, r'+1, r'-1,
i'+1, i'-1, A'[m']:=A[i'],
A'[i'] := A[m'], N[m']
}

P = {
m'<m, r'>0, A'[i']=A'[i'+1]
}

Example of templates

In-place run-length encoding:

A = [1,1,1,0,0,2,2,2,2]

Original encoder

A = [1,0,2]

N = [3,2,4]

Template decoder

A′ = [1,1,1,0,0,2,2,2,2]

```
assume(n>=0);
i, m := 0, 0;       // par
while (i < n)
  r := 1;
  while (i+1 < n && A
    r, i := r + 1, i+1;
  A[m], N[m], m, i := A
```

Template control flow, expressions E, and predicates P, semi-automatically mined from original

```
i', m' := e1, e2;   // ei ∈ E
while (p1)          // pi ∈ P
  r' := e3;
  while (p2)
    r', i', A' := e4, e5, e6;
  m' := e7;
```

E = {
0, 1, m'+1, m'-1, r'+1, r'-1, i'+1, i'-1, A'[m']:=A[i'], A'[i'] := A[m'], N[m']
}

P = {
m'<m, r'>0, A'[i']=A'[i'+1]
}

Symbolic execution of program paths

```
assume(n>=0);
i, m := 0, 0;
while (i < n)
  r := 1;
  while (i+1 < n && A[i] = A[i+1])
    r, i := r + 1, i+1;
  A[m], N[m], m, i := A[i], r, m+1, i+1;
```

```
i', m' := e1, e2;
while (p1)
  r' := e3;
  while (p2)
    r', i', A' := e4, e5, e6;
  m' := e7;
```

Symbolic execution of program paths

```
assume(n>=0);
i, m := 0, 0;
while (i < n)
    r := 1;
    while (i+1 < n && A[i] = A[i+1])
        r, i := r + 1, i+1;
    A[m], N[m], m, i := A[i], r, m+1, i+1;
```

```
i', m' := e1, e2;
while (p1)
    r' := e3;
    while (p2)
        r', i', A' := e4, e5, e6;
    m' := e7;
```

Symbolic execution of program paths

assume(n>=0);
i, m := 0, 0;
while (i < n)
  r := 1;
  while (i+1 < n && A[i] = A[i+1])
    r, i := r + 1, i+1;
  A[m], N[m], m, i := A[i], r, m+1, i+1;

$(n^0 >= 0) \wedge$
$i^1 = 0 \wedge m^1 = 0 \wedge$
$\neg\ (i^1 < n^0) \wedge$

$i'^1 = e_1^V \wedge m'^1 = e_2^V \wedge$
$\neg\ (p_1^V)$
$\Rightarrow$ identity

i’, m’ := e_1, e_2;
while (p_1)
  r’ := e_3;
  while (p_2)
    r’, i’, A’ := e_4, e_5, e_6;
  m’ := e_7;

(n>=0)
i, m := 0, 0
$\neg$ (i < n)

i’, m’ := e_1, e_2
$\neg$ (p_1)

Symbolic execution of program paths

```
assume(n>=0);
i, m := 0, 0;
while (i < n)
  r := 1;
  while (i+1 < n && A[i] = A[i+1])
    r, i := r + 1, i+1;
  A[m], N[m], m, i := A[i], r, m+1, i+1;
```

$(n^0 >= 0) \wedge$

$i^1 = 0 \wedge m^1 = 0 \wedge$

$\neg\ (i^1 < n^0) \wedge$

```
i', m' := e1, e2;
while (p1)
  r' := e3;
  while (p2)
    r', i', A' := e4, e5, e6;
  m' := e7;
```

$i'^1 = e_1^V \wedge m'^1 = e_2^V \wedge$

$\neg\ (p_1^V)$

$\Rightarrow$ identity

Symbolic execution of program paths

```
assume(n>=0);
i, m := 0, 0;
while (i < n)
    r := 1;
    while (i+1 < n && A[i] = A[i+1])
        r, i := r + 1, i+1;
    A[m], N[m], m, i := A[i], r, m+1, i+1;
```

```
i', m' := e1, e2;
while (p1)
    r' := e3;
    while (p2)
        r', i', A' := e4, e5, e6;
    m' := e7;
```

$(n^0 >= 0) \wedge$

$i^1 = 0 \wedge m^1 = 0 \wedge$

$\neg\ (i^1 < n^0) \wedge$

$i'^1 = e_1^V \wedge m'^1 = e_2^V \wedge$

$\neg\ (p_1^V)$

$\Rightarrow$ identity

Symbolic execution of program paths

```
assume(n>=0);
i, m := 0, 0;
while (i < n)
    r := 1;
    while (i+1 < n && A[i] = A[i+1])
        r, i := r + 1, i+1;
    A[m], N[m], m, i := A[i], r, m+1, i+1;
```

$(n^0 >= 0) \wedge$
$i^1 = 0 \wedge m^1 = 0 \wedge$
$\neg\ (i^1 < n^0) \wedge$

$i'^1 = e_1^V \wedge m'^1 = e_2^V \wedge$
$\neg\ (p_1^V)$
$\Rightarrow$ identity

$(n^0 >= 0) \wedge$
$i^1 = 0 \wedge m^1 = 0 \wedge$
$\neg\ (i^1 < n^0) \wedge$

$i'^1 = e_1^V \wedge m'^1 = e_2^V \wedge$
$(p_1^{V'}) \wedge$
$r'^1 = e_3^{V'} \wedge$
$\neg\ (p_2^{V''}) \wedge$
$m'^2 = e_7^{V''} \wedge$
$\neg\ (p_1^{V'''})$ $\Rightarrow$ identity

```
i', m' := e1, e2;
while (p1)
    r' := e3;
    while (p2)
        r', i', A' := e4, e5, e6;
    m' := e7;
```

```
   (n>=0)
   i, m := 0, 0
¬ (i < n)

   i', m' := e1, e2
   (p1)
   r' := e3
¬ (p2)
   m' := e7
¬ (p1)
```

Symbolic execution of program paths

```
assume(n>=0);
i, m := 0, 0;
while (i < n)
  r := 1;
  while (i+1 < n && A[i] = A[i+1])
    r, i := r + 1, i+1;
  A[m], N[m], m, i := A[i], r, m+1, i+1;
```

```
i', m' := e1, e2;
while (p1)
  r' := e3;
  while (p2)
    r', i', A' := e4, e5, e6;
  m' := e7;
```

$(n^0 >= 0) \wedge$
$i^1 = 0 \wedge m^1 = 0 \wedge$
$\neg\ (i^1 < n^0) \wedge$

$i'^1 = e_1^V \wedge m'^1 = e_2^V \wedge$
$\neg\ (p_1^V)$
$\Rightarrow$ identity

$(n^0 >= 0) \wedge$
$i^1 = 0 \wedge m^1 = 0 \wedge$
$\neg\ (i^1 < n^0) \wedge$

$i'^1 = e_1^V \wedge m'^1 = e_2^V \wedge$
$(p_1^{V'}) \wedge$
$r'^1 = e_3^{V'} \wedge$
$\neg\ (p_2^{V''}) \wedge$
$m'^2 = e_7^{V''} \wedge$
$\neg\ (p_1^{V'''})$ $\Rightarrow$ identity

Symbolic execution of program paths

```
assume(n>=0);
i, m := 0, 0;
while (i < n)
  r := 1;
  while (i+1 < n && A[i] = A[i+1])
    r, i := r + 1, i+1;
  A[m], N[m], m, i := A[i], r, m+1, i+1;
```

```
i', m' := e1, e2;
while (p1)
  r' := e3;
  while (p2)
    r', i', A' := e4, e5, e6;
  m' := e7;
```

$(n^0 >= 0) \wedge$
$i^1 = 0 \wedge m^1 = 0 \wedge$
$\neg\ (i^1 < n^0) \wedge$

$i'^1 = e_1^V \wedge m'^1 = e_2^V \wedge$
$\neg\ (p_1^V)$
$\Rightarrow$ identity

$(n^0 >= 0) \wedge$
$i^1 = 0 \wedge m^1 = 0 \wedge$
$\neg\ (i^1 < n^0) \wedge$

$i'^1 = e_1^V \wedge m'^1 = e_2^V \wedge$
$(p_1^{V'}) \wedge$
$r'^1 = e_3^{V'} \wedge$
$\neg\ (p_2^{V''}) \wedge$
$m'^2 = e_7^{V''} \wedge$
$\neg\ (p_1^{V'''})$ $\Rightarrow$ identity

Symbolic execution of program paths

```
assume(n>=0);
i, m := 0, 0;
while (i < n)
  r := 1;
  while (i+1 < n && A[i] = A[i+1])
    r, i := r + 1, i+1;
  A[m], N[m], m, i := A[i], r, m+1, i+1;
```

$(n^0 >= 0) \wedge$
$i^1 = 0 \wedge m^1 = 0 \wedge$
$\neg\ (i^1 < n^0) \wedge$

$i'^1 = e_1{}^V \wedge m'^1 = e_2{}^V \wedge$
$\neg\ (p_1{}^V)$
$\Rightarrow$ identity

$(n^0 >= 0) \wedge$
$i^1 = 0 \wedge m^1 = 0 \wedge$
$\neg\ (i^1 < n^0) \wedge$

$i'^1 = e_1{}^V \wedge m'^1 = e_2{}^V \wedge$
$(p_1{}^{V'}) \wedge$
$r'^1 = e_3{}^{V'} \wedge$
$\neg\ (p_2{}^{V''}) \wedge$
$m'^2 = e_7{}^{V''} \wedge$
$\neg\ (p_1{}^{V'''})$ $\Rightarrow$ identity

```
i', m' := e1, e2;
while (p1)
  r' := e3;
  while (p2)
    r', i', A' := e4, e5, e6;
  m' := e7;
```

```
  (n>=0)
  i, m := 0, 0
  (i < n)
  r := 1
¬ (i+1 < n && A[i] = A[i+1])
  A[m], N[m], m, i := A[i], r, m+1, i+1
¬ (i < n)
  i', m' := e1, e2
¬ (p1)
```

Symbolic execution of program paths

```
assume(n>=0);
i, m := 0, 0;
while (i < n)
  r := 1;
  while (i+1 < n && A[i] = A[i+1])
    r, i := r + 1, i+1;
  A[m], N[m], m, i := A[i], r, m+1, i+1;
```

$(n^0 >= 0) \wedge$
$i^1 = 0 \wedge m^1 = 0 \wedge$
$\neg\ (i^1 < n^0) \wedge$

$i'^1 = e_1^V \wedge m'^1 = e_2^V \wedge$
$\neg\ (p_1^V)$
$\Rightarrow$ identity

$(n^0 >= 0) \wedge$
$i^1 = 0 \wedge m^1 = 0 \wedge$
$\neg\ (i^1 < n^0) \wedge$

$i'^1 = e_1^V \wedge m'^1 = e_2^V \wedge$
$(p_1^{V'}) \wedge$
$r'^1 = e_3^{V'} \wedge$
$\neg\ (p_2^{V''}) \wedge$
$m'^2 = e_7^{V''} \wedge$
$\neg\ (p_1^{V'''})$ $\Rightarrow$ identity

```
i', m' := e1, e2;
while (p1)
  r' := e3;
  while (p2)
    r', i', A' := e4, e5, e6;
  m' := e7;
```

```
  (n>=0)
  i, m := 0, 0
  (i < n)
  r := 1
¬ (i+1 < n && A[i] = A[i+1])
  A[m], N[m], m, i := A[i], r, m+1, i+1
¬ (i < n)
  i', m' := e1, e2
¬ (p1)
```

$(n^0 >= 0) \wedge$
$i^1 = 0 \wedge m^1 = 0 \wedge$
$(i^1 < n^0) \wedge$
$r^1 = 1 \wedge$
$\neg\ (i^1+1 < n^0 \ \&\& \ A[i^1] = A[i^1+1])$
$A[m^1]=A[i^1] \wedge N[m^1]=r^1 \wedge m^2= m^1+1 \wedge i^2=i^1+1$
$\neg\ (i^2 < n^0)$
$i'^1 = e_1^V \wedge m'^1 = e_2^V \wedge$
$\neg\ (p_1^V)$
$\Rightarrow$ identity

Symbolic execution of program paths

$(n^0>=0) \wedge$
$i^1 = 0 \wedge m^1 = 0 \wedge$
$\neg (i^1 < n^0) \wedge$

$i'^1=e_1^V \wedge m'^1=e_2^V \wedge$
$\neg (p_1^V)$
$\Rightarrow$ identity

$(n^0>=0) \wedge$
$i^1 = 0 \wedge m^1 = 0 \wedge$
$\neg (i^1 < n^0) \wedge$

$i'^1=e_1^V \wedge m'^1=e_2^V \wedge$
$(p_1^{V'}) \wedge$
$r'^1=e_3^{V'} \wedge$
$\neg (p_2^{V''}) \wedge$
$m'^2=e_7^{V''} \wedge$
$\neg (p_1^{V'''})$ $\Rightarrow$ identity

$(n^0>=0) \wedge$
$i^1 = 0 \wedge m^1 = 0 \wedge$
$(i^1 < n^0) \wedge$
$r^1 = 1 \wedge$
$\neg (i^1+1 < n^0 \;\&\&\; A[i^1] = A[i^1+1])$
$A[m^1]=A[i^1] \wedge N[m^1]=r^1 \wedge m^2= m^1+1 \wedge i^2=i^1+1$
$\neg (i^2 < n^0)$
$i'^1=e_1^V \wedge m'^1=e_2^V \wedge$
$\neg (p_1^V)$
$\Rightarrow$ identity

Solving using SMT and SAT

$(n^0 >= 0) \wedge$
$i^1 = 0 \wedge m^1 = 0 \wedge$
$\neg\ (i^1 < n^0) \wedge$

$i'^1 = e_1^V \wedge m'^1 = e_2^V \wedge$
$\neg\ (p_1^V)$
$\Rightarrow$ identity

$(n^0 >= 0) \wedge$
$i^1 = 0 \wedge m^1 = 0 \wedge$
$\neg\ (i^1 < n^0) \wedge$

$i'^1 = e_1^V \wedge m'^1 = e_2^V \wedge$
$(p_1^{V'}) \wedge$
$r'^1 = e_3^{V'} \wedge$
$\neg\ (p_2^{V''}) \wedge$
$m'^2 = e_7^{V''} \wedge$
$\neg\ (p_1^{V'''})$ $\Rightarrow$ identity

$(n^0 >= 0) \wedge$
$i^1 = 0 \wedge m^1 = 0 \wedge$
$(i^1 < n^0) \wedge$
$r^1 = 1 \wedge$
$\neg\ (i^1+1 < n^0\ \&\&\ A[i^1] = A[i^1+1])$
$A[m^1]=A[i^1] \wedge N[m^1]=r^1 \wedge m^2= m^1+1 \wedge i^2=i^1+1$
$\neg\ (i^2 < n^0)$
$i'^1 = e_1^V \wedge m'^1 = e_2^V \wedge$ $\Rightarrow$ identity
$\neg\ (p_1^V)$

Solving using SMT and SAT

$\varphi_1(e_1,e_2,p_1)$
$\Rightarrow$ identity

$(n^0 >= 0) \wedge$
$i^1 = 0 \wedge m^1 = 0 \wedge$
$\neg\ (i^1 < n^0) \wedge$

$i'^1 = e_1^V \wedge m'^1 = e_2^V \wedge$
$(p_1^{V'}) \wedge$
$r'^1 = e_3^{V'} \wedge$
$\neg\ (p_2^{V''}) \wedge$
$m'^2 = e_7^{V''} \wedge$
$\neg\ (p_1^{V'''})$ $\Rightarrow$ identity

$(n^0 >= 0) \wedge$
$i^1 = 0 \wedge m^1 = 0 \wedge$
$(i^1 < n^0) \wedge$
$r^1 = 1 \wedge$
$\neg\ (i^1+1 < n^0\ \&\&\ A[i^1] = A[i^1+1])$
$A[m^1]=A[i^1] \wedge N[m^1]=r^1 \wedge m^2= m^1+1 \wedge i^2=i^1+1$
$\neg\ (i^2 < n^0)$
$i'^1 = e_1^V \wedge m'^1 = e_2^V \wedge$ $\Rightarrow$ identity
$\neg\ (p_1^V)$

Solving using SMT and SAT

$\varphi_1(e_1, e_2, p_1)$
$\Rightarrow$ identity

$\varphi_2(e_1, e_2, p_1, e_3, e_7, p_2)$
$\Rightarrow$ identity

$(n^0 >= 0) \wedge$
$i^1 = 0 \wedge m^1 = 0 \wedge$
$(i^1 < n^0) \wedge$
$r^1 = 1 \wedge$
$\neg\ (i^1+1 < n^0\ \&\&\ A[i^1] = A[i^1+1])$
$A[m^1]=A[i^1] \wedge N[m^1]=r^1 \wedge m^2= m^1+1 \wedge i^2=i^1+1$
$\neg\ (i^2 < n^0)$
$i'^1=e_1^V \wedge m'^1=e_2^V \wedge$ $\Rightarrow$ identity
$\neg\ (p_1^V)$

Solving using SMT and SAT

$\varphi_1(e_1,e_2,p_1)$
⇒ identity

$\varphi_2(e_1,e_2,p_1,e_3,e_7,p_2)$
⇒ identity

$\varphi_2(e_1,e_2,p_1)$
⇒ identity

Solving using SMT and SAT

$\exists\, e_i, p_j\ \forall$ program vars

$\varphi_1(e_1, e_2, p_1)$
$\Rightarrow$ identity

$\wedge$

$\varphi_2(e_1, e_2, p_1, e_3, e_7, p_2)$
$\Rightarrow$ identity

$\varphi_2(e_1, e_2, p_1)$
$\Rightarrow$ identity

Solving using SMT and SAT

$\exists\, e_i, p_j\ \forall$ program vars

$\varphi_1(e_1, e_2, p_1) \Rightarrow$ identity

$\wedge$

$\varphi_2(e_1, e_2, p_1, e_3, e_7, p_2) \Rightarrow$ identity

$\varphi_2(e_1, e_2, p_1) \Rightarrow$ identity

- Naive approach:
  - Enumerate e_i, p_j and "validate"
  - Will not scale
    - 2^{11} to 2^{37} candidates our experiments

Efficient solving from prior work on verification using SAT/SMT

$$\exists e_i, p_j \forall vars \; \bigwedge_k \varphi_k(e_1, e_2, p_1) \Rightarrow identity$$

- Efficient solving strategy:
  - Verification solves $\exists Invariant \forall vars$
  - Reuse SMT-based verifier technology
- Core idea:
  - Predicates/expressions form a lattice
  - Efficient encoding using lattice instead of enumerating entire domain
  - See prior work in PLDI'09/POPL'10

The PINS Algorithm

C = termination (T)
while (true) {
  solns = solve (C,P,E,Spec)
  if (empty(**solns**)) fail
  if (stabilized(**solns**)) return **solns**
  s = pickone (**solns**)
  C = **C**∧ directed-path-explore (T,**s**)
}

The PINS Algorithm

C holds the accumulated constraints

C = termination (T)
while (true) {
  solns = solve (C,P,E,Spec)
  if (empty(**solns**)) fail
  if (stabilized(**solns**)) return **solns**
  s = pickone (**solns**)
  C = **C**∧ directed-path-explore (T,**s**)
}

The PINS Algorithm

C holds the accumulated constraints

Initialize with termination cnstr →

(Simple linear constraints that ensure that symbolic execution terminates)

```
C = termination (T)
while (true) {
   solns = solve (C,P,E,Spec)
   if (empty(solns)) fail
   if (stabilized(solns)) return solns
   s = pickone (solns)
   C = C∧ directed-path-explore (T,s)
}
```

The PINS Algorithm

C holds the accumulated constraints

Initialize with termination cnstr →

(Simple linear constraints that ensure that symbolic execution terminates)

If no change to candidate set then they are likely not refutable →

```
C = termination (T)
while (true) {
   solns = solve (C,P,E,Spec)
   if (empty(solns)) fail
   if (stabilized(solns)) return solns
   s = pickone (solns)
   C = C∧ directed-path-explore (T,s)
}
```

The PINS Algorithm

C holds the accumulated constraints

Initialize with termination cnstr →

(Simple linear constraints that ensure that symbolic execution terminates)

If no change to candidate set then they are likely not refutable

Else use one **s** to parameterize next path exploration

C = termination (T)
while (true) {
  solns = solve (C,P,E,Spec)
  if (empty(**solns**)) fail
  if (stabilized(**solns**)) return **solns**
  s = pickone (**solns**)
  C = **C**∧ directed-path-explore (T,**s**)
}

The PINS Algorithm

C holds the accumulated constraints

Initialize with termination cnstr →

(Simple linear constraints that ensure that symbolic execution terminates)

If no change to candidate set then they are likely not refutable

Else use one **s** to parameterize next path exploration

Explore another path and add its constraint

C = termination (T)
while (true) {
  solns = solve (C,P,E,Spec)
  if (empty(**solns**)) fail
  if (stabilized(**solns**)) return **solns**
  s = pickone (**solns**)
  C = **C**∧ directed-path-explore (T,**s**)
}

The PINS Algorithm

We do not have a way of certifiably saying which remaining solutions are correct and which are not.

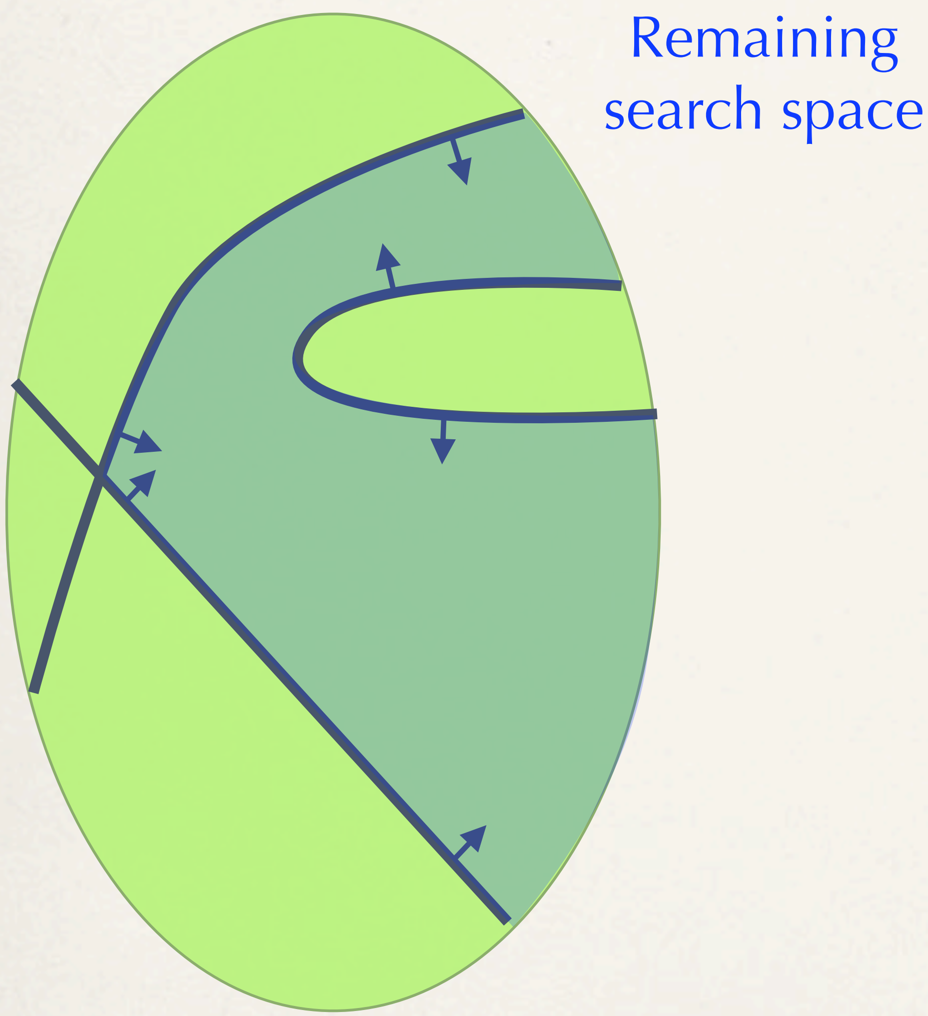
So how do we find a path that prunes the space further?

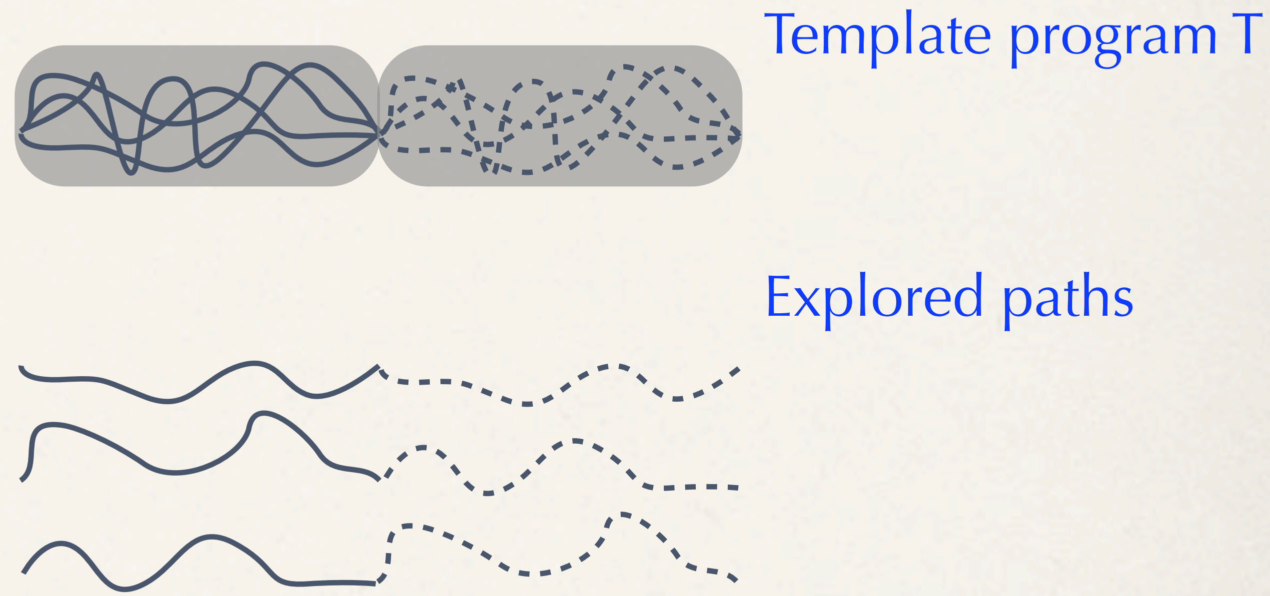
Else use one **s** to parameterize next path exploration

Explore another path and add its constraint

C = termination (T)
while (true) {
  solns = solve (C,P,E,Spec)
  if (empty(**solns**)) fail
  if (stabilized(**solns**)) return **solns**
  s = pickone (**solns**)
  C = **C**∧ directed-path-explore (T,**s**)
}

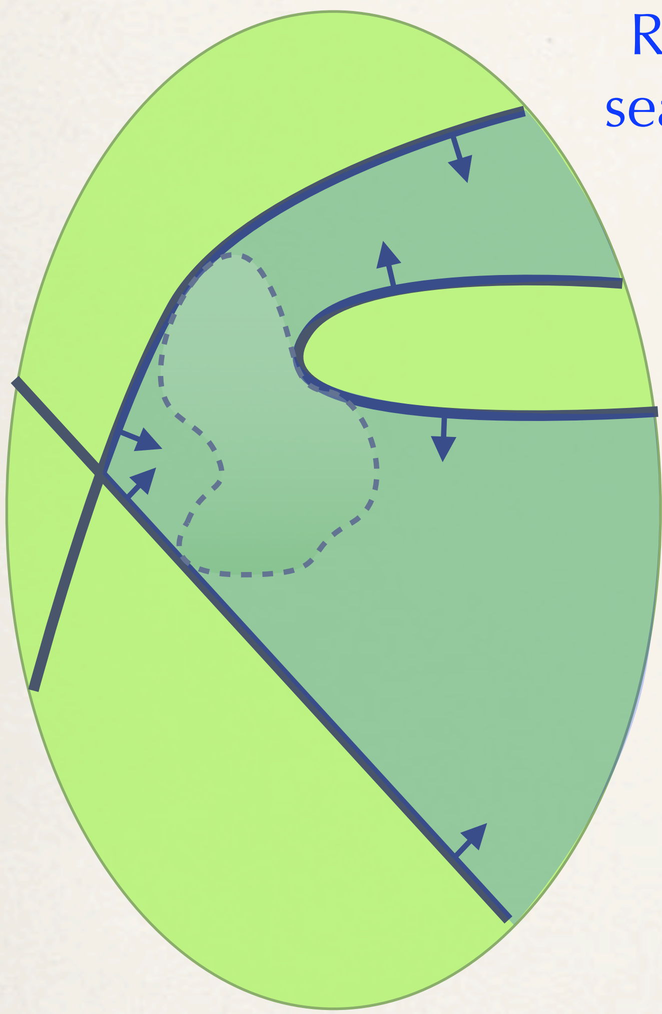
Directed path exploration



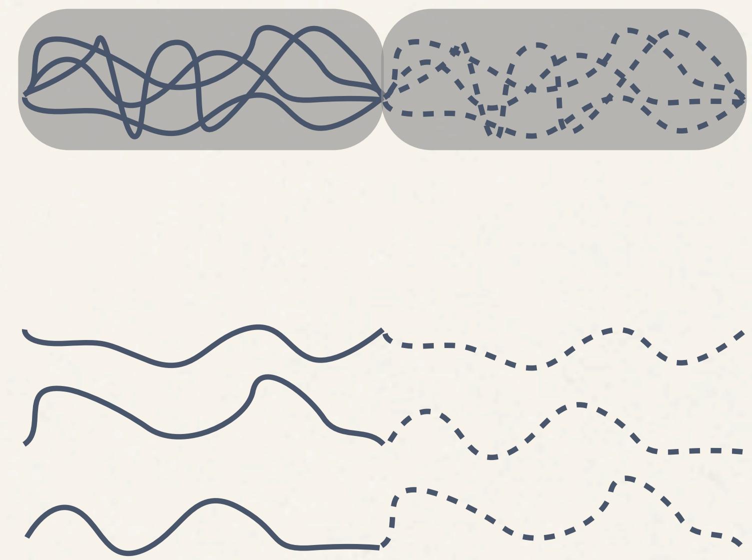


$2^{|P|^{(\# \text{ Pred Holes})}} \times |E|^{(\# \text{ Expr Holes})}$

Directed path exploration



Remaining search space

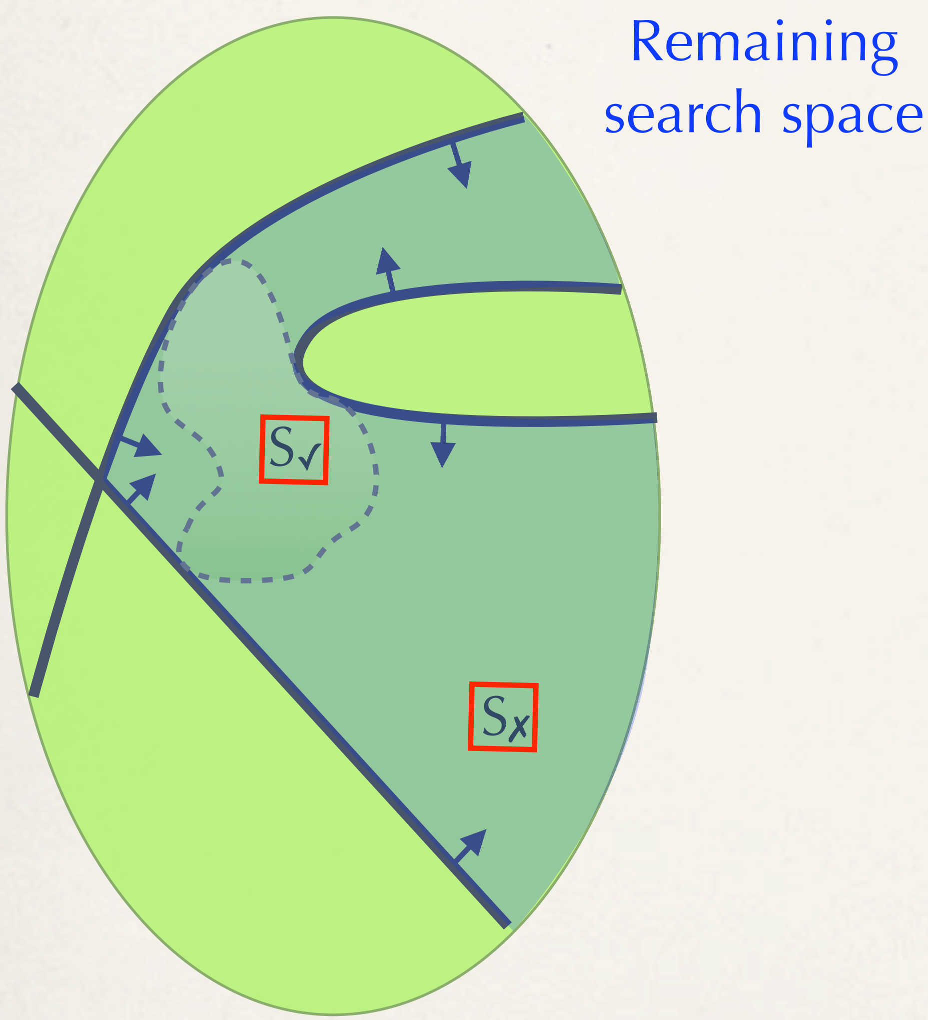


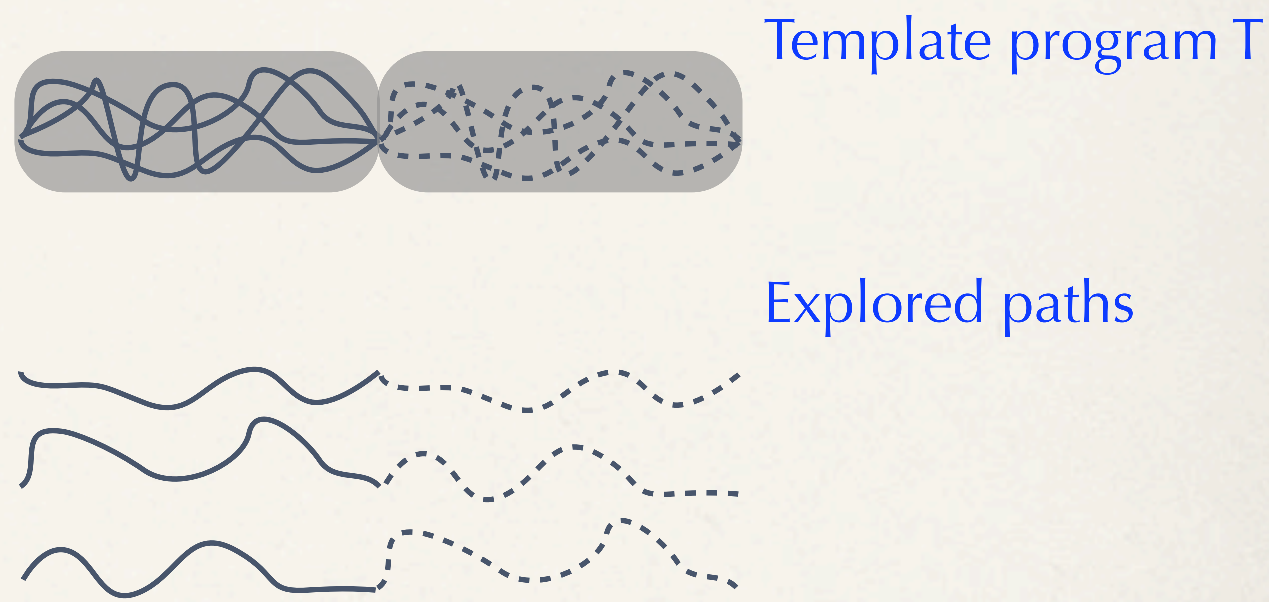
Template program T

Explored paths

$$2^{|P|^{(\#\ \text{Pred Holes})}} \times |E|^{(\#\ \text{Expr Holes})}$$

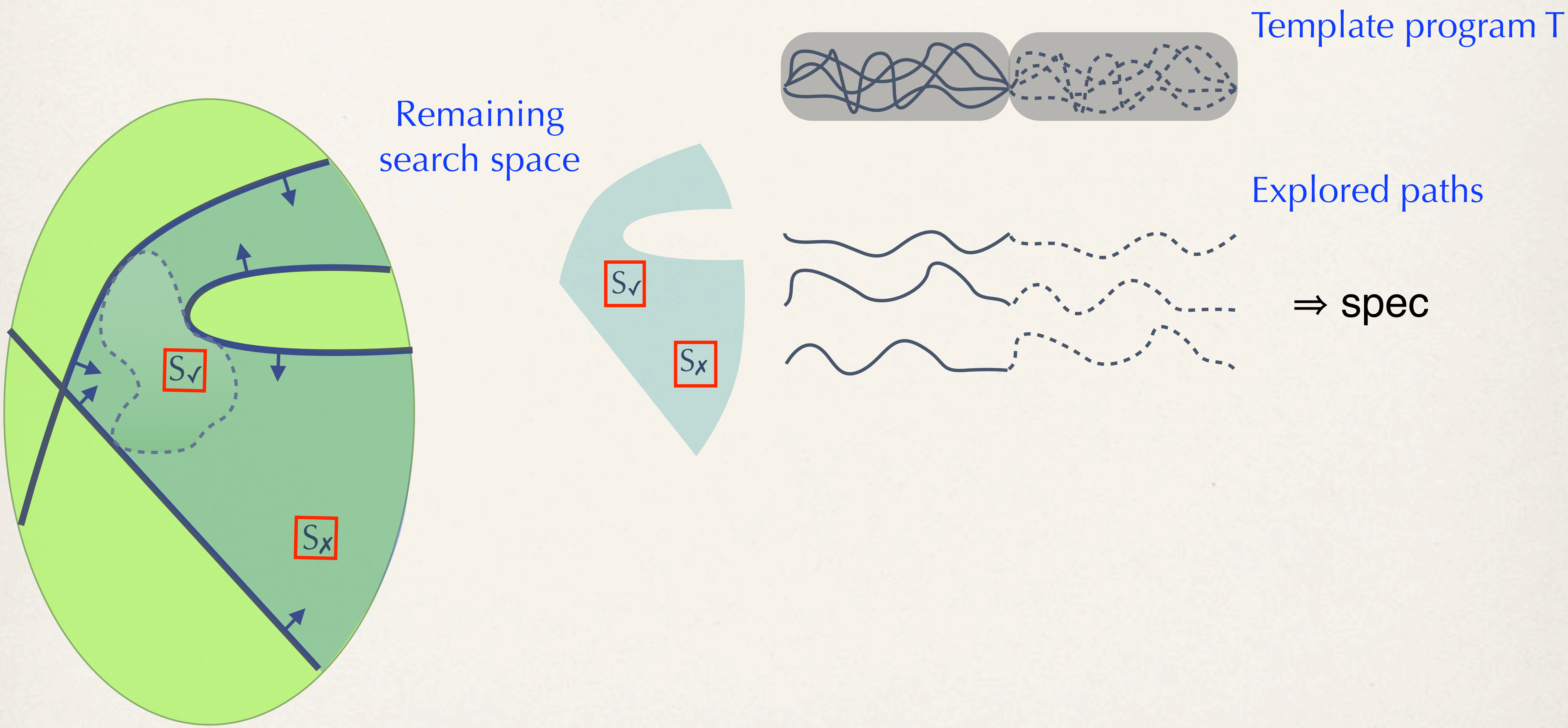
Directed path exploration





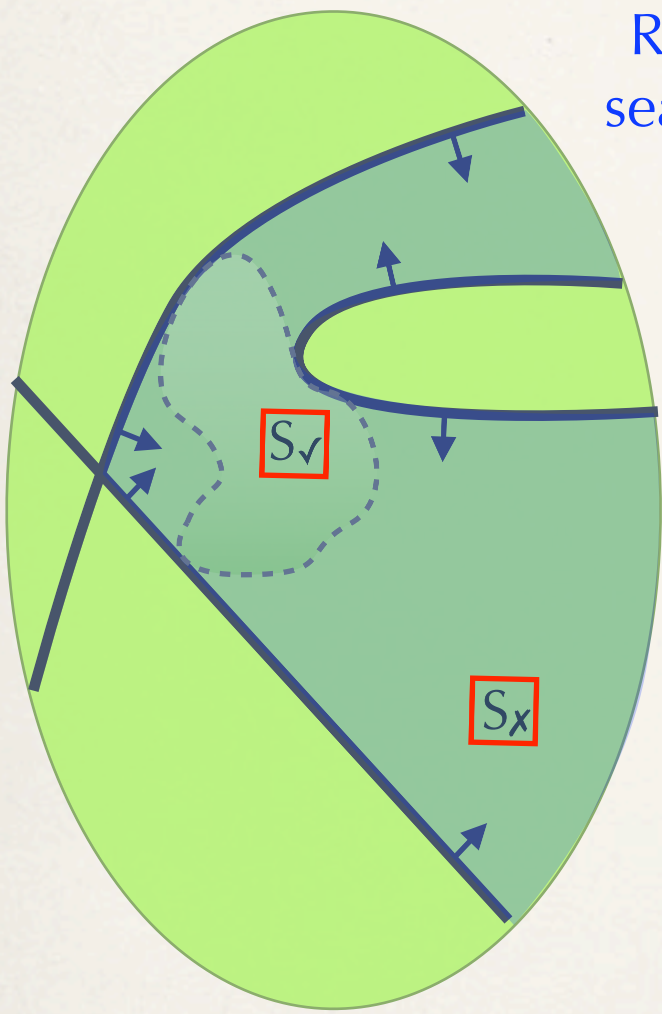
$2^{|P|^{(\#\text{ Pred Holes})}} \times |E|^{(\#\text{ Expr Holes})}$

Directed path exploration

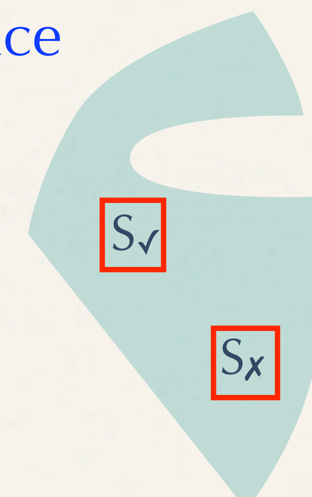


$2^{|P|^{(\text{\# Pred Holes})}} \times |E|^{(\text{\# Expr Holes})}$

Directed path exploration

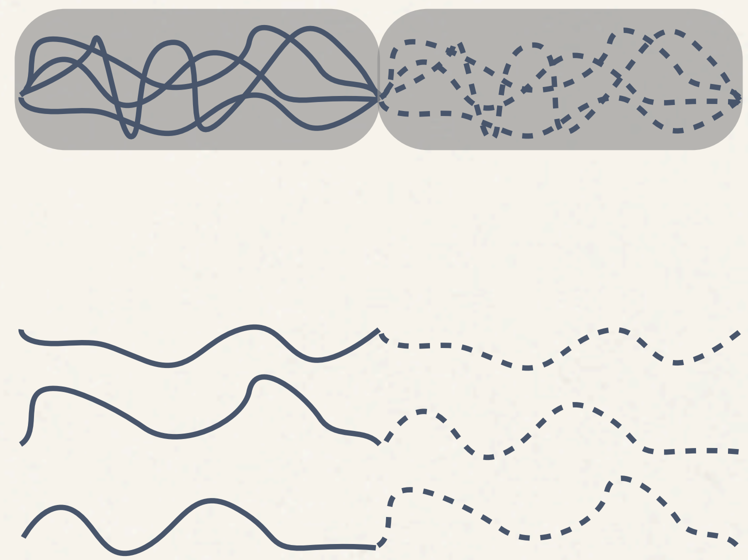


Remaining search space



What if?

Template program T

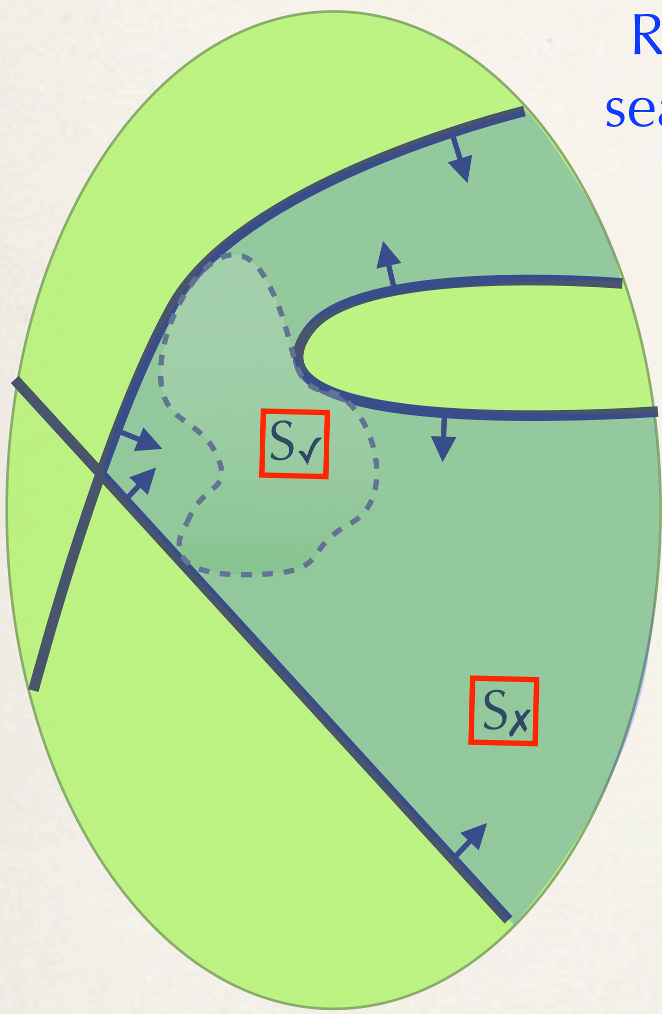


Explored paths

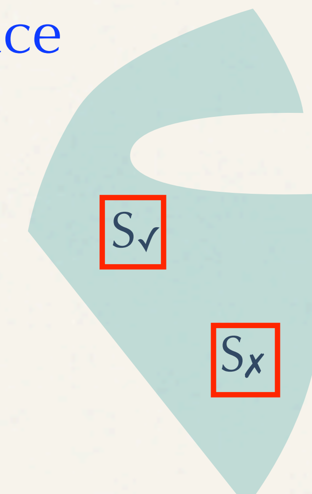
⇒ spec

$2^{|P|^{(\# \text{ Pred Holes})}} \times |E|^{(\# \text{ Expr Holes})}$

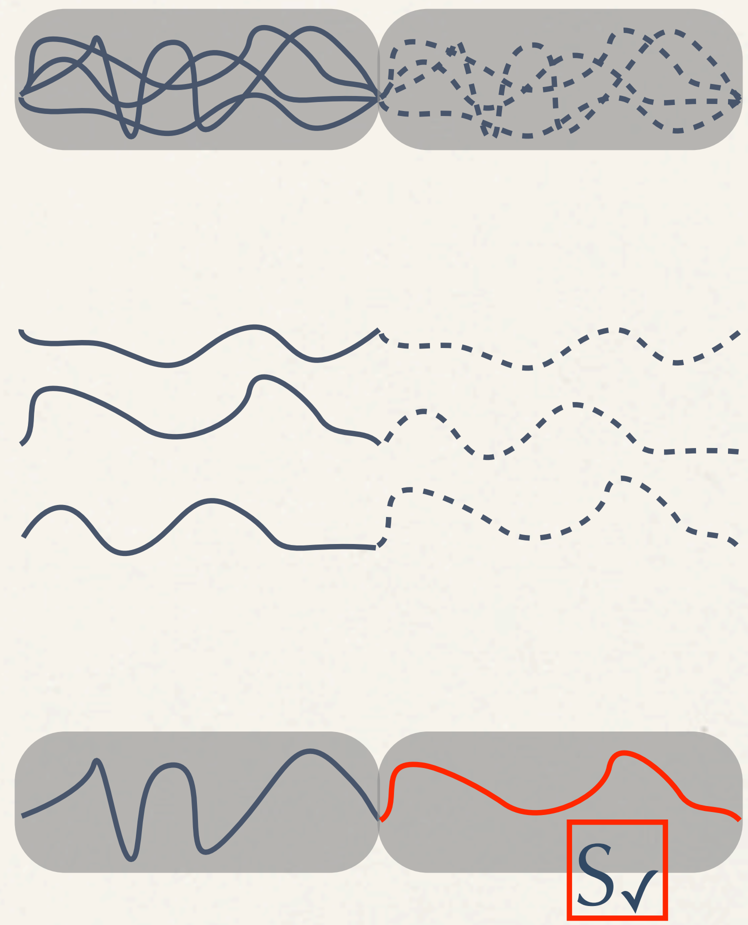
Directed path exploration



Remaining search space



Template program T



Explored paths

⇒ spec

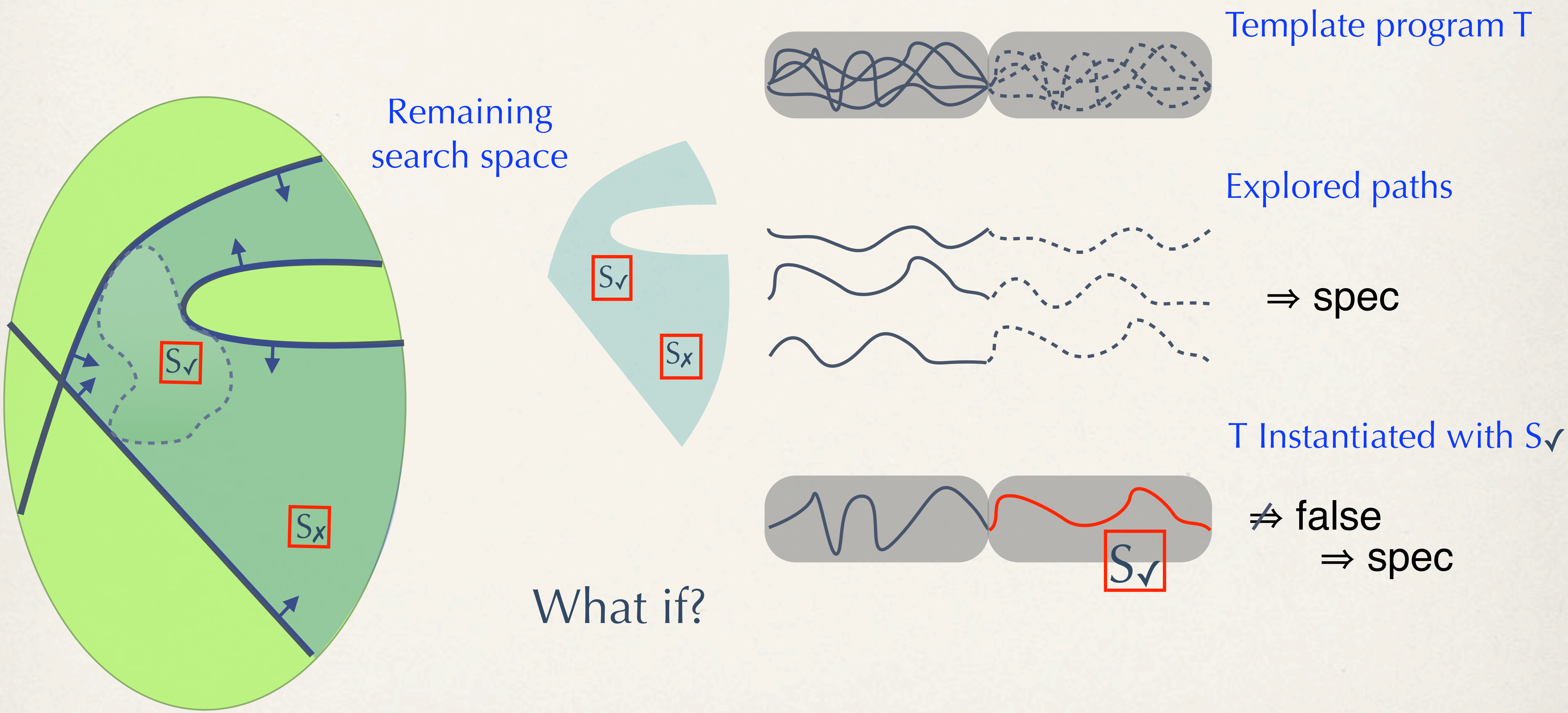
T Instantiated with S✓

⇏ false

What if?

$2^{|P|^{(\# \text{ Pred Holes})}} \times |E|^{(\# \text{ Expr Holes})}$

Directed path exploration

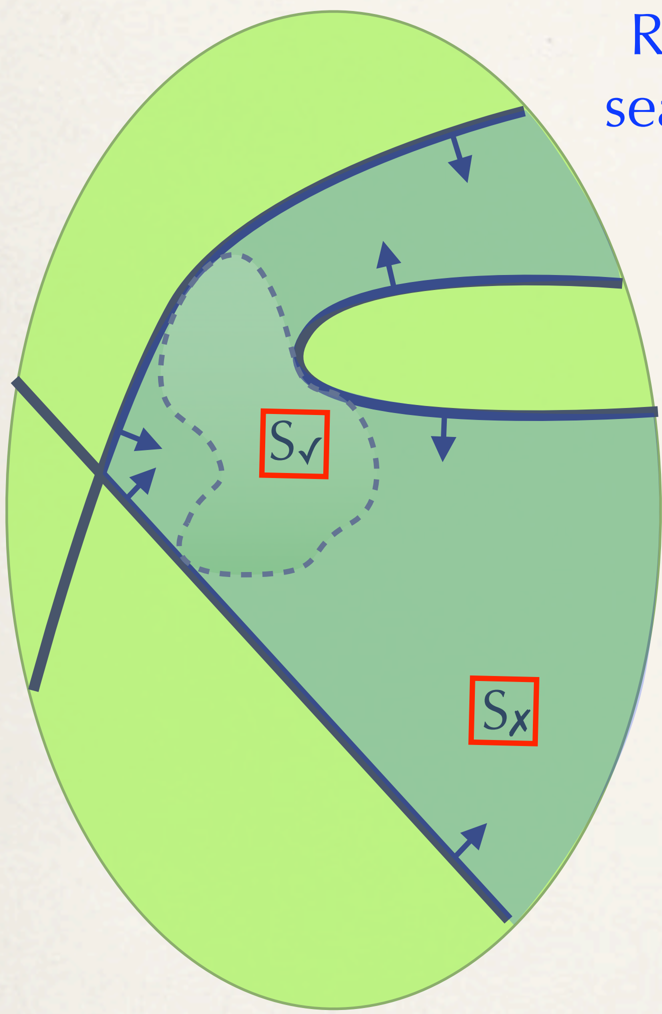


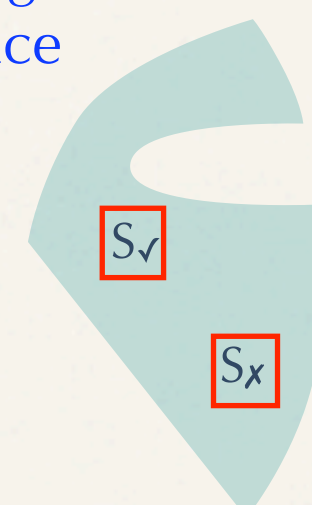
What if?

$2^{|P|^{(\#\text{ Pred Holes})}} \times |E|^{(\#\text{ Expr Holes})}$

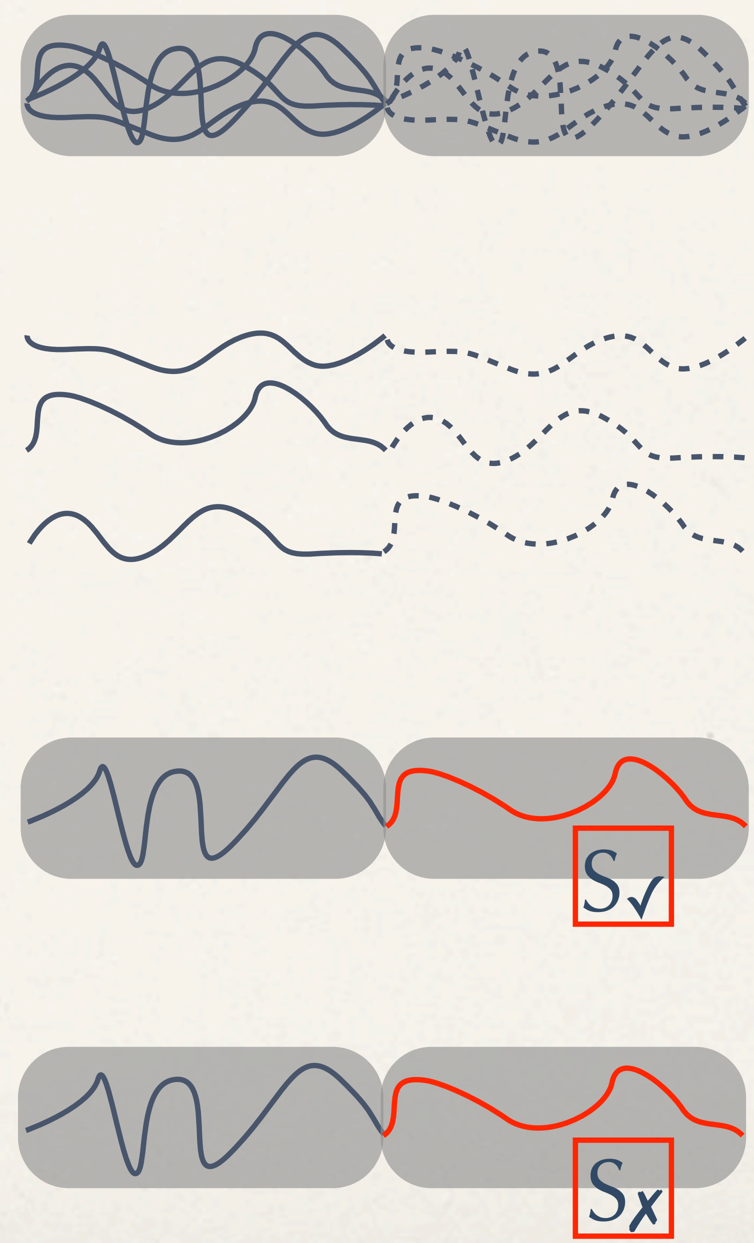
Directed path exploration

Remaining search space



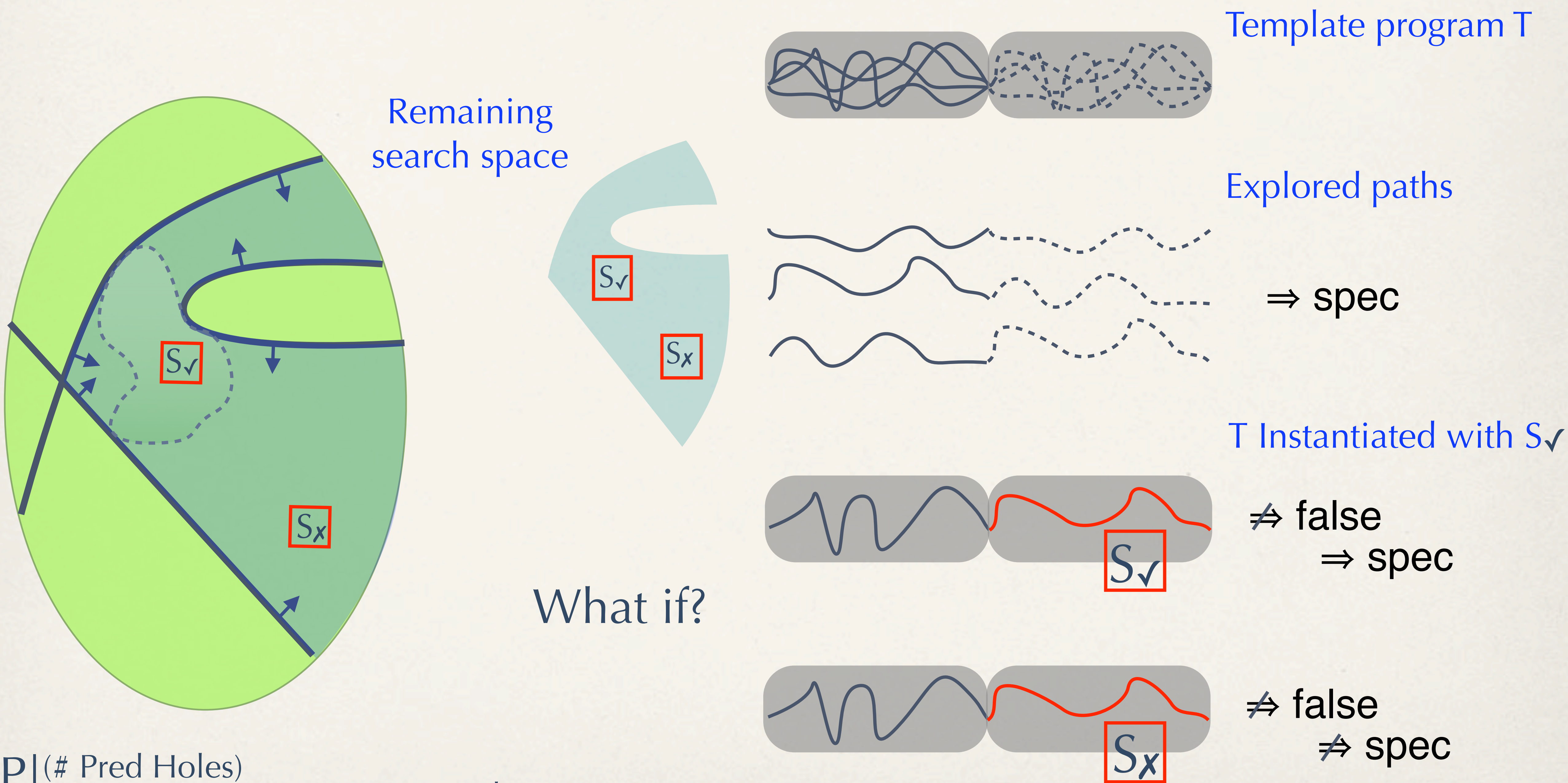


What if?



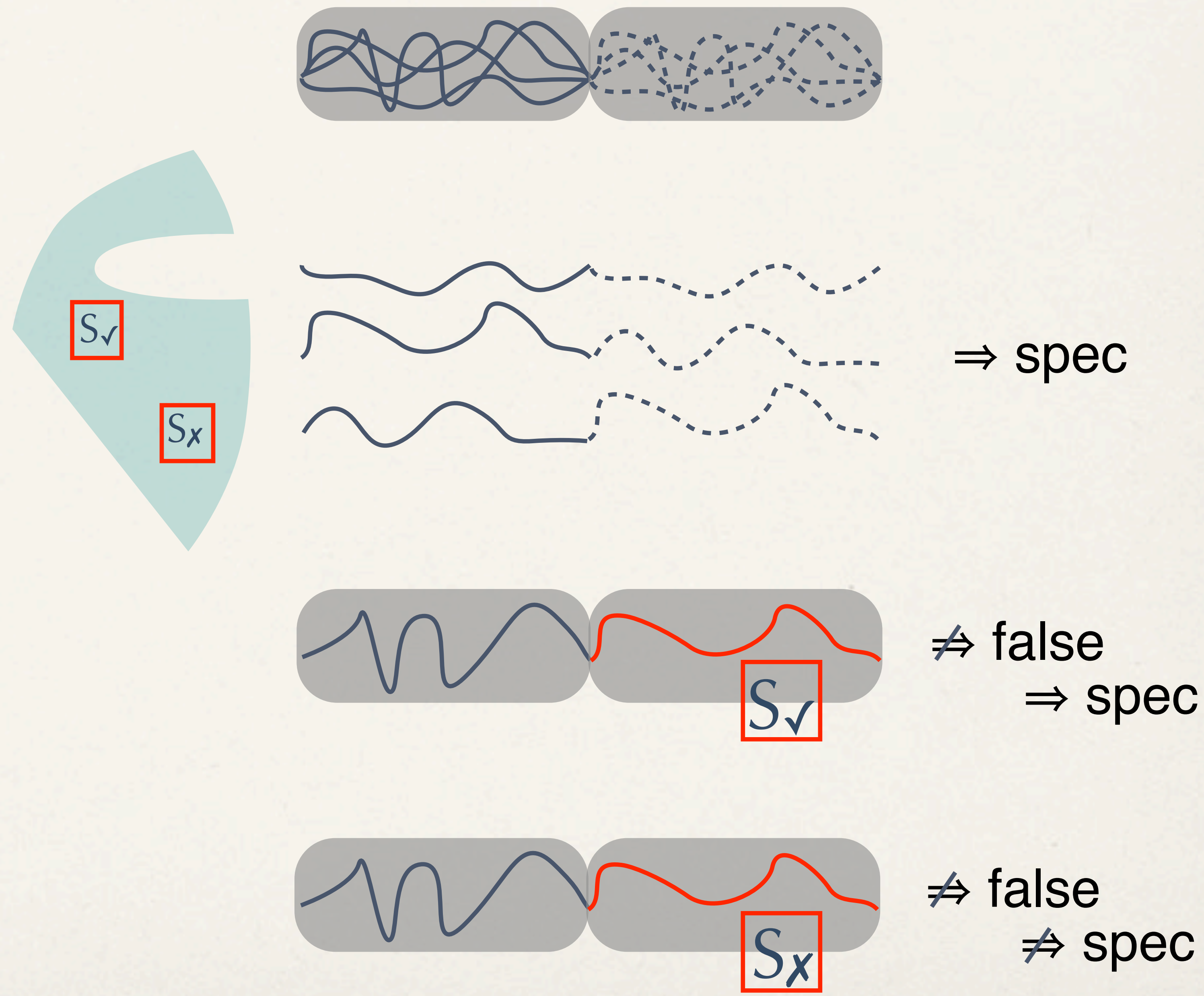
$2^{|P|^{(\text{\# Pred Holes})}} \times |E|^{(\text{\# Expr Holes})}$

Directed path exploration



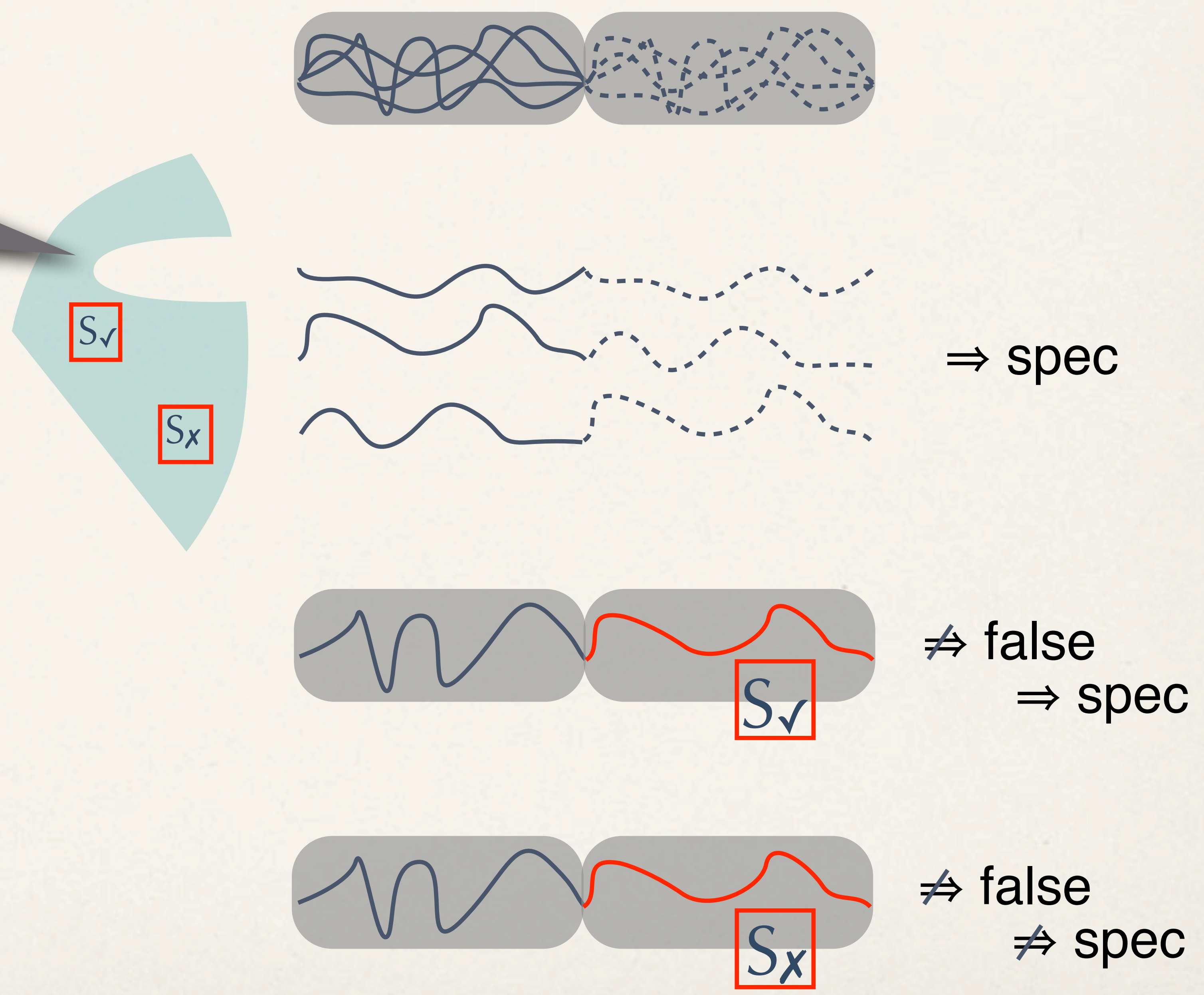
$2^{|P|^{(\# \text{ Pred Holes})}} \times |E|^{(\# \text{ Expr Holes})}$

Directed path exploration



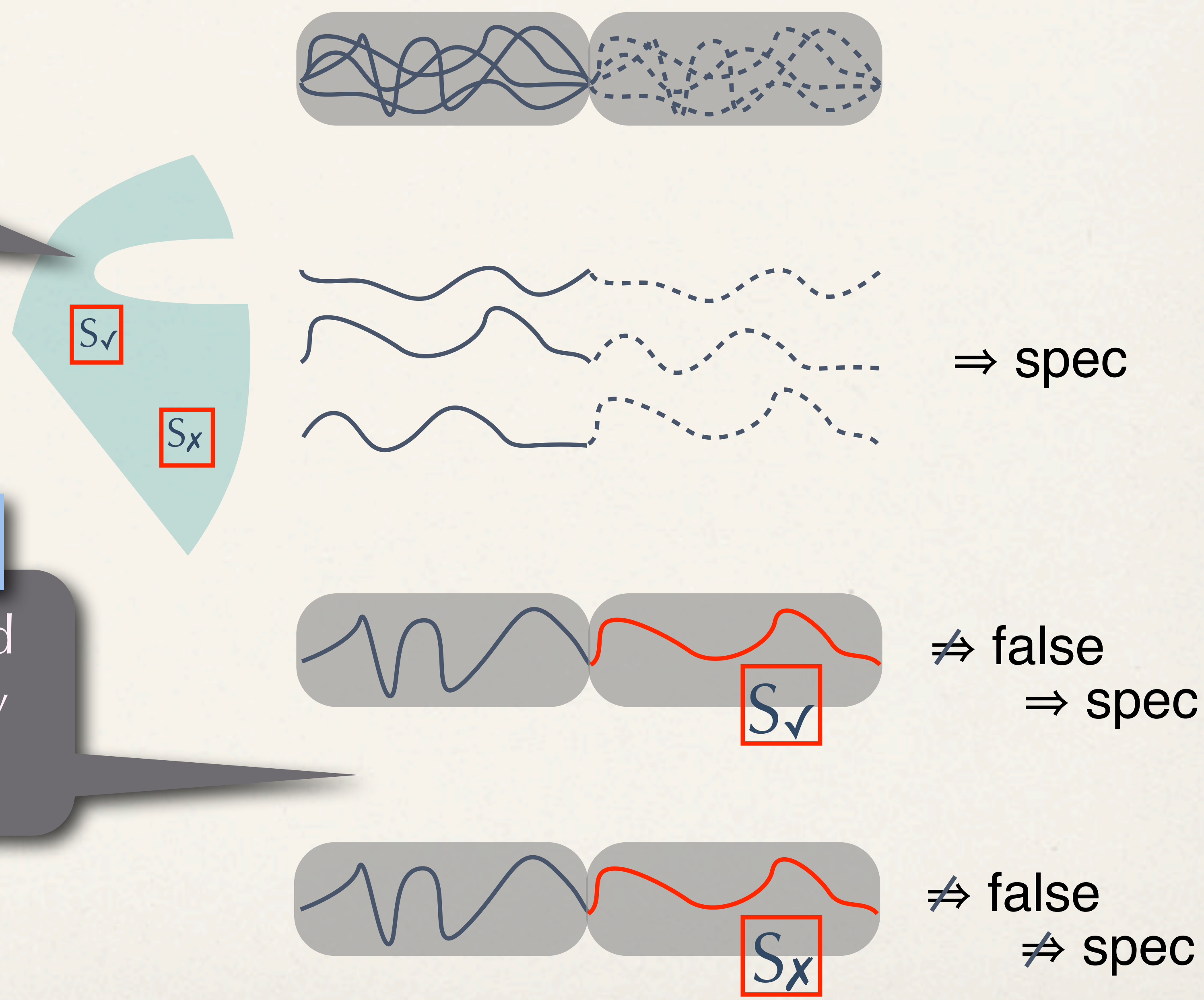
Directed path exploration

Pick any solution from remaining space; don't care about its validity



Directed path exploration

Pick any solution from remaining space; don't care about its validity



Directed path exploration

Instantiate template with picked solution, and now symbolically execute to find *feasible* path

Program inversion benchmarks

- Three domains
  - Lossless compression
  - Format conversion
  - Arithmetic
- Semi-automatic procedure to extract template T
  - Control-flow derived from original program
  - Expression/predicates mined
- Ran PINS to invert using template T

Results

	Benchmark	Search space reduction	Number of Iterations	Time	Manual Check	Model Checker
Lossless Compression	Run length	2^{25} -> 1	7	26s	ok	validated
	In place RL	2^{30} -> 1	7	36s	ok	validated
	LZ77	2^{25} -> 2	6	1810s	1 of 2 ok	validated
	LZW	2^{31} -> 2	4	150s	2 of 2 ok	too complex
Format Conversion	Base 64	2^{37} -> 4	12	1377s	1 of 4 ok	too complex
	UUencode	2^{20} -> 1	7	34s	ok	too complex
	Pkt Wrap	2^{20} -> 1	6	132s	ok	too complex
	Serialize	2^{11} -> 1	14	55s	ok	too complex
Arithmetic	Sum i	2^{15} -> 1	4	1s	ok	validated
	Vector rotate	2^{16} -> 1	3	4s	ok	too complex
	Vector shift	2^{16} -> 1	3	4s	ok	validated
	Vector scale	2^{16} -> 1	3	36s	ok	too complex

Results

Category	Benchmark	Search space reduction	Number of Iterations	Time	Manual Check	Model Checker
Lossless Compression	Run length	2^{25} -> 1	7	26s	ok	validated
	In place RL	2^{30} -> 1	7	36s	ok	validated
	LZ77	2^{25} -> 2	[illegible]	[illegible]	1 of 2 ok	validated
	LZW	2^{31} -> 2	[illegible]	[illegible]	2 of 2 ok	too complex
Format Conversion	Base 64	2^{37} -> 4	[illegible]	[illegible]	1 of 4 ok	too complex
	UUencode	2^{20} -> 1	7	34s	ok	too complex
	Pkt Wrap	2^{20} -> 1	6	132s	ok	too complex
	Serialize	2^{11} -> 1	14	55s	ok	too complex
Arithmetic	Sum i	2^{15} -> 1	4	1s	ok	validated
	Vector rotate	2^{16} -> 1	3	4s	ok	too complex
	Vector shift	2^{16} -> 1	3	4s	ok	validated
	Vector scale	2^{16} -> 1	3	36s	ok	too complex

PINS narrowed the valid candidates to 1 in almost all cases

Results

Category	Benchmark	Search space reduction	Number of Iterations	Time	Manual Check	Model Checker
Lossless Compression	Run length	2^{25} -> 1	7	26s	ok	validated
	In place RL	2^{30} -> 1	7	36s	ok	validated
	LZ77	2^{25} -> 2	6	[illegible]	[illegible]	validated
	LZW	2^{31} -> 2	4	[illegible]	[illegible]	too complex
Format Conversion	Base 64	2^{37} -> 4	12	[illegible]	[illegible]	too complex
	UUencode	2^{20} -> 1	7	[illegible]	[illegible]	too complex
	Pkt Wrap	2^{20} -> 1	6	132s	ok	too complex
	Serialize	2^{11} -> 1	14	55s	ok	too complex
Arithmetic	Sum i	2^{15} -> 1	4	1s	ok	validated
	Vector rotate	2^{16} -> 1	3	4s	ok	too complex
	Vector shift	2^{16} -> 1	3	4s	ok	validated
	Vector scale	2^{16} -> 1	3	36s	ok	too complex

Directed path exploration is successful in finding a small set of paths that prune the space

Results

Benchmark	Search space reduction	Number of Iterations	Time	Manual Check	Model Checker
Lossless Compression					
Run length	2^{25} -> 1	7	26s	ok	validated
In place RL	2^{30} -> 1	7	36s	ok	validated
LZ77	2^{25} -> 2	6	1810s	1 of 2 ok	validated
LZW	2^{31} -> 2	4	150s	2 of 2 ok	too complex
Format Conversion					
Base 64	2^{37} -> 4	12	1377s	1 of 4 ok	too complex
UUencode	2^{20} -> 1	7	34s	ok	too complex
Pkt Wrap	2^{20} -> 1	6	132s	ok	too complex
Serialize	2^{11} -> 1	14	55s	[illegible]	…mplex
Arithmetic					
Sum i	2^{15} -> 1	4	1s	[illegible]	[illegible]
Vector rotate	2^{16} -> 1	3	4s	[illegible]	…x
Vector shift	2^{16} -> 1	3	4s	[illegible]	…dated
Vector scale	2^{16} -> 1	3	36s	ok	too complex

Symbolic execution is sometimes expensive; but mostly the paths are explored in reasonable time

Results

	Benchmark	Search space reduction	Number of Iterations	Time	Manual Check	Model Checker
Lossless Compression	Run length	2^{25} -> 1	7	26s	ok	validated
	In place RL	2^{30} -> 1	7	36s	ok	validated
	LZ77	2^{25} -> 2	6	1810s	1 of 2 ok	validated
	LZW	2^{31} -> 2	4	150s	2 of 2 ok	too complex
Format Conversion	Base 64	2^{37} -> 4	12	1377s	1 of 4 ok	too complex
	UUencode	2^{20} -> 1	7	34s	ok	too complex
	Pkt Wrap	2^{20} -> 1	6	[illegible]	ok	too complex
	Serialize	2^{11} -> 1	[illegible]	[illegible]	ok	too complex
Arithmetic	Sum i	2^{15} -> 1	4	[illegible]	ok	validated
	Vector rotate	2^{16} -> 1	3	[illegible]	ok	too complex
	Vector shift	2^{16} -> 1	3	4s	ok	validated
	Vector scale	2^{16} -> 1	3	36s	ok	too complex

Either only one remained or were easily examined

Results

	Benchmark	Search space reduction	Number of Iterations	Time	Manual Check	Model Checker
Lossless Compression	Run length	2^{25} -> 1	7	26s	ok	validated
	In place RL	2^{30} -> 1	7	36s	ok	validated
	LZ77	2^{25} -> 2	6	1810s	1 of 2 ok	validated
	LZW	2^{31} -> 2	4	150s	2 of 2 ok	too complex
Format Conversion	Base 64	2^{37} -> 4	12	1377s	1 of 4 ok	too complex
	UUencode	2^{20} -> 1	7	34s	ok	too complex
	Pkt Wrap	2^{20} -> 1	6	132s	ok	too complex
	Serialize	2^{11} -> 1	14	[illegible]	[illegible]	too complex
Arithmetic	Sum i	2^{15} -> 1	[illegible]	[illegible]	[illegible]	validated
	Vector rotate	2^{16} -> 1	[illegible]	[illegible]	[illegible]	too complex
	Vector shift	2^{16} -> 1	3	[illegible]	ok	validated
	Vector scale	2^{16} -> 1	3	36s	ok	too complex

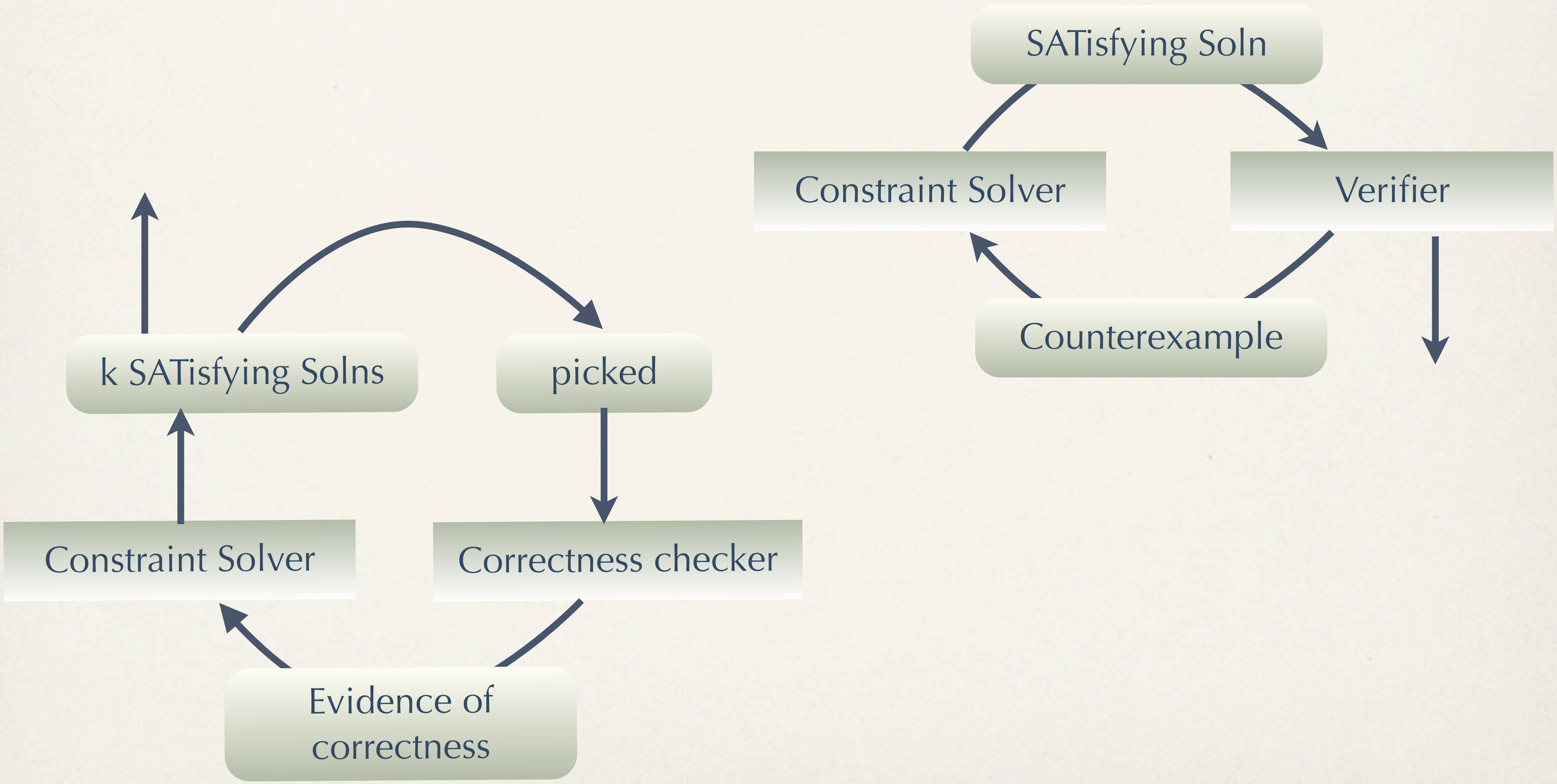
A testing-based approach is the only viable option, as most of the examples are too complex for even for bounded verification

Conclusions

- PINS seems very promising
  - First testing-based approach to program synthesis
  - To our knowledge, no other technique can invert these programs with as little guidance
- Supports small path-bound hypothesis for synthesis
  - Makes sense, since it works for testing (approximate verification), and we know verification and synthesis are related (see POPL'10 paper)
- PINS should be applicable to other domains too

http://www.cs.umd.edu/~saurabhs/vs3/PINS/

PINS approach vs CEGAR/CEGIS



LZW

```
void main(int *A, int n) {
  int *P,*N,*C;
  int i,j,k,c,p,r;

   IN(BOUND(A,0,n),n);
   ASSUME(n >= 0);
   i = 0; k = 0;
   while (i < n) {
      c = 0; p = 0; j = 0;
      while (j < i) {
         r = 0;
         while (i+r < n-1  && A[j+r] == A[i+r])
            r++;
         if (c < r) {
            c = r; p = i-j;
         }
         j++;
      }
      P[k] = p; N[k] = c; C[k] = A[i+c];
      i = i+1+c;
      k++;
   }
   OUT(P,N,C,k);
}
```

LZW compressor

PINS

```
void main(int n, BitString A) {
   BitString *D;
   int *B;
   int i,p,k,j,r,size,x,go;

   IN(str(A,0,n-1),n);
   ASSUME(n >= 1);

   D[0] = "0";
   D[1] = "1";

   i = 0; p = 2; k = 0;
   while (i < n) {
      j = i; r = 0; size=-1;
      while (j < n && r != -1) {
         x = 0; r = -1;
         while (x < p) {
            if (D[x] == substr(A,i,j))
               r = x;
            x++;
         }
         if (r != -1)
         {  go = r; size = j-i+1; }
         j++;
      }
      B[k++] = go;
      D[p++] = substr(A,i,j-1);
      i += size;
   }

   OUT(B,k);
}
```

LZW decompressor