SPoC: Search-based Pseudocode to Code

Sumith Kulal*, Panupong Pasupat*, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, Percy Liang
Department of Computer Science, Stanford University

Summary: **Given pseudocode and test cases, the task is to synthesize a program, which will be evaluated on functional correctness. We release the SPoC dataset (18K programs + human-authored pseudocode), a search-based synthesizer, and error localization models to guide search based on information from compilation errors.**

Task and Motivation

Goal: Synthesize programs that are *long* (10–20 lines) and *functionally correct*.

```
 i  x_i                                      y_i
 1  in function main                        int main() {
 2    let n be integer                        int n;
 3    read n                                  cin >> n;
 4    let A be vector of integers             vector<int> A;
 5    set size of A = n                       A.resize(n);
 6    read n elements into A                  for(int i = 0; i < A.size(); i++) cin >> A[i];
 7    for all elements in A                   for(int i = 0; i < A.size(); i++) {
 8      set min_i to i                          int min_i = i;
 9      for j = i + 1 to size of A exclusive    for(int j = i+1; j < A.size(); j++) {
10        set min_i to j if A[min_i] > A[j]       if(A[min_i] > A[j]) { min_i = j; }
11      swap A[i], A[min_i]                     swap(A[i], A[min_i]);
12    print all elements of A                 for(int i=0; i<A.size(); i++) cout<<A[i]<<" ";
                                            }

          Public test case 1 (out of 5):  5 3 2 4 1 5        → 1 2 3 4 5
          Hidden test case 1 (out of 8):  8 9 2 4 5 6 2 7 1 → 1 2 2 4 5 6 7 9
```

- **Input:** pseudocode lines $x_{1:N}$ + public test cases
- **Output:** a program with code lines $y_{1:N}$
- **Evaluation:**
  - **Functional correctness:** The program must pass both public + private test cases.
  - **Performance metric:** number of synthesis trials
    (1 trial = 1 compiler call + execution on all public test cases)

Why? Most existing works either generate short programs or ignore functional correctness during evaluation.

	Input	Output size	Evaluate functional correctness?
Semantic parsing	natural language	usually short (e.g., SQL, logical forms)	yes (e.g., database execution)
Language to code	natural language	long (e.g., methods, classes)	mostly no (e.g., exact match, BLEU)
Test-driven program synthesis	test cases	usually short	yes
This work	natural language + test cases	long (program)	yes

SPoC Dataset

bit.ly/spoc-dataset

Main features:

- Complex programs from programming competitions + test cases, inspired by the NAPS dataset (Zavershynskyi et al., NAMPI 2018).
- 18356 programs
- All programs come with **human-authored pseudocode**.

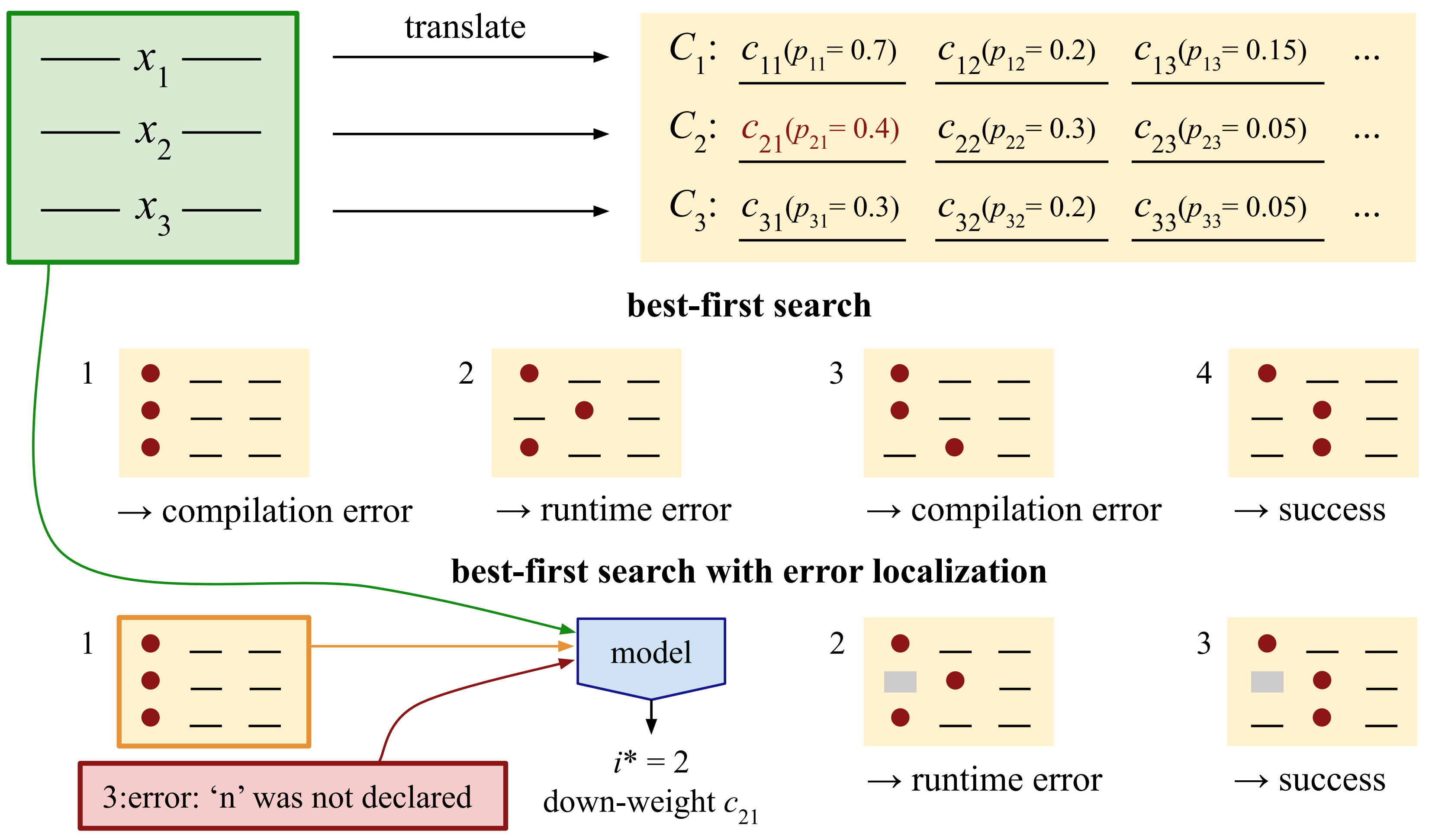
Local-level challenges: Translating each line is non-trivial.

High-level descriptions	
read n values into array a and array b	`for(int i = 0; i < n; i++) cin >> a[i] >> b[i];`
read n and m in a loop, printing … n*m/2 and a new line on each iteration	`while (cin >> n >> m) cout << n * m / 2 << endl;`
print all elements of ans	`for (int i = 0; i < ans.size(); i++) cout << ans[i];`
Complex sentences and diverse operations	
change max to i if tree[i] > tree[max] … or max otherwise	`max = tree[i] > tree[max] ? i : max;`
if m and n are odd	`if (m % 2 != 0 && n % 2 != 0)`
if a is a digit return 1	`if (a >= '0' && a <= '9') return 1;`
Context-dependent interpretation	
add s to q (q is a set)	`q.insert(s);`
add ok to ans (ans is an integer)	`ans += ok;`
add element a to v (v is a vector)	`v.push_back(a);`

Global-level challenges:

- The programs are 14 lines long on average.
- One wrong code line can make the whole program incorrect!
- And most programs have at least one difficult line. (See the experiments)

Two data splits: TestP (split by problem) and TestW (split by pseudocode author).



Base Framework

Step 1: Translate. For each pseudocode line x_i, use a standard seq2seq model to generate candidate code lines c_{ij} with probability $p_{ij} = p(c_{ij}|x_i)$.

Step 2: Best first search. Start from the top prediction ($y_i = c_{i1}$). Iterate through possible combinations $y_i = c_{ij}$ with decreasing joint probability $\prod_i p_{ij}$ until the program passes all public test cases (or the budget exhausts).

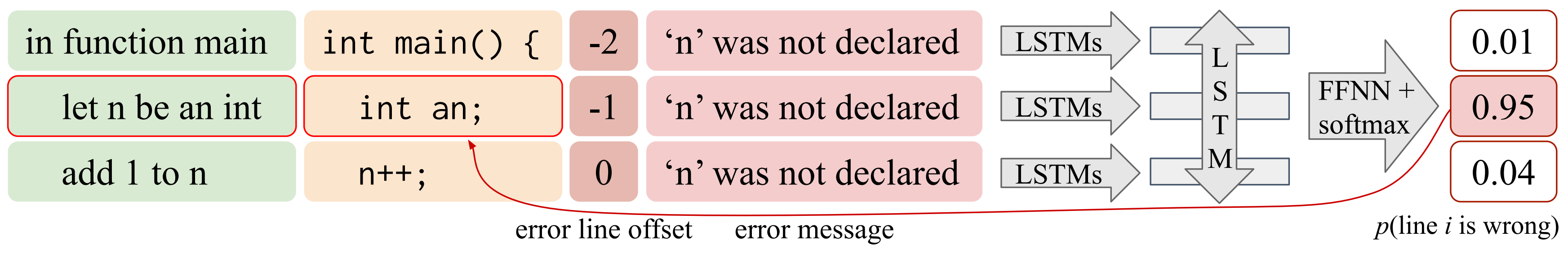
Error Localization

When a program fails, we want to avoid using the source of that failure over again.

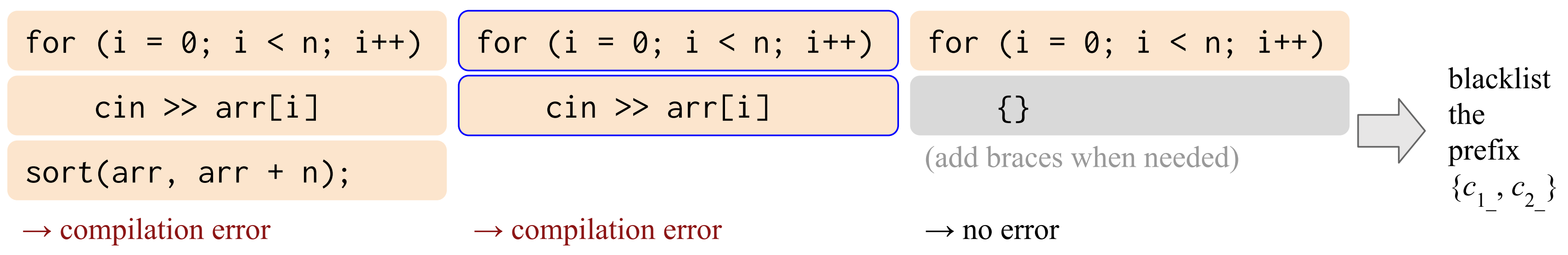
Proposal: When a compilation error occurs, use an *error localization method* to infer the offending code line(s), then demote or blacklist them.

In the example figure above, (c_{11}, c_{22}, c_{32}) satisfies the test cases. Best-first search iterates in the order of decreasing probabilities and succeeds in 4 compiler calls. The error localization method down-weights c_{21}, leading to an earlier success.

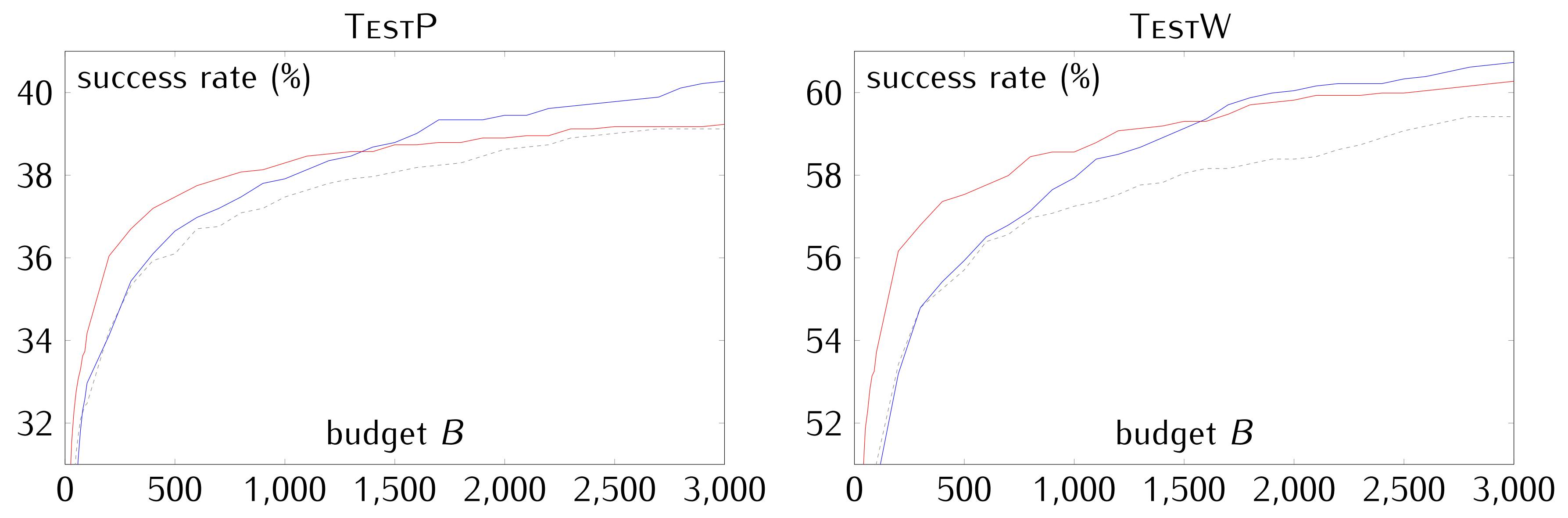
Method 1: Multiclass classification. Predict the offending line from the error message.



Method 2: Prefix-based pruning. Spend a few trials to validate code prefixes.



Experiments and Takeaways



TestP:

$B =$	10	100	1000	3000
no localization	26.5	32.5	37.5	39.1
multiclass	**28.4**	**34.2**	**38.3**	39.2
prefix-based	25.3	33.0	37.9	**40.3**

top-one ($B = 1$): 17.8 oracle ($B = \infty$): 55.2

TestW:

$B =$	10	100	1000	3000
no localization	42.5	51.0	57.3	59.4
multiclass	**44.4**	**53.7**	**58.6**	60.3
prefix-based	41.0	50.5	57.9	**60.7**

top-one ($B = 1$): 30.7 oracle ($B = \infty$): 71.4

Takeaway 1: Long programs → more chances to go wrong. Even though line-level translation accuracy is 85%, stitching the top translations gives a success rate of 24.6%.

Takeaway 2: Search increases the success rate. Under the budget of 100 trials, the success rate goes up to 44.7%.

Takeaway 3: Error localization reduces the number of trials needed:

- The multiclass classification model reduces the number of trials needed in 15.5% of the programs (median reduction of 26 trials).
- Prefix-based pruning increases the number of trials on easy problems (since we need to compile prefixes) but greatly helps on harder programs.