香港科技大學
THE HONG KONG
UNIVERSITY OF SCIENCE
AND TECHNOLOGY

UNIVERSITY OF
OXFORD

Parameterized Algorithms and Complexity for Function Merging with Branch Reordering

Amir Goharshady, Kerim Kochekov, Tian Shu, Ahmed Zaher

Jun 19th, 2026

Code size optimization

Compilers usually optimize for **speed**..

Code size optimization

Compilers usually optimize for **speed**..

.. we focus on optimizing for **size**.

Code size optimization

Compilers usually optimize for **speed**..

.. we focus on optimizing for **size**.

Applications: mobile apps, embedded systems.

Function merging

Function Merging

Input: Two functions F_1 and F_2.

Output: Merged function $F_\star$ such that:

- $F_\star$ captures both F_1 and F_2, and
- $F_\star$ is as small as possible.

Function merging (ex.)

Simplifying assumption: size $=$ # lines.

Function merging (ex.)

```
1  def F1(x):
2    x += 1
3    y = x * 75
4    z = -2 * x
5    print(y)
6    if x == 0:
7      print(x)
8      for i in range(3):
9        print(i**2)
10   elif x == 1:
11     print(x * z)
12   print(x**2)
```

```
13 def F2(x):
14   x += 1
15   z = -2 * x
16   print(x + z)
17   if x == 1:
18     print(x * z)
19   elif x == 0:
20     print(x)
21     for i in range(3):
22       print(i**2)
23   print(x**2)
```

Function merging (ex.)

```
1  def F⋆(x, b):
2    x += 1
3    if b == 1:
4      y = x * 75
5    z = -2 * x
6    if b == 1:
7      print(y)
8    else:
9      print(x + z)
10   if b == 2 and x == 1:
11     print(x * z)
12   elif x == 0:
13     print(x)
14     for i in range(3):
15       print(i**2)
16   elif b == 1 and x == 1:
17     print(x * z)
18   print(x**2)
```

```
19 def F1(x):
20   F⋆(x, 1)
```

```
21 def F2(x):
22   F⋆(x, 2)
```

Function merging (ex.)

23 lines $\rightarrow$ 22 lines!

Function merging via sequence alignment

SOTA approach: **sequence alignment** [CGO'19, PLDI'20].

Function merging via sequence alignment

SOTA approach: **sequence alignment** [CGO'19, PLDI'20].

- View F_1 and F_2 as **strings of instructions**.
- **Align** some identical pairs of instructions accross F_1/F_2.

Function merging via sequence alignment

SOTA approach: **sequence alignment** [CGO'19, PLDI'20].

- View F_1 and F_2 as **strings of instructions**.
- **Align** some identical pairs of instructions accross F_1/F_2.
- *Aligned pair of instr.*: merged; always executed.
- *Unaligned inst.:* executed conditionally.

Function merging via sequence alignment

SOTA approach: **sequence alignment** [CGO'19, PLDI'20].

- View F_1 and F_2 as **strings of instructions**.
- **Align** some identical pairs of instructions accross F_1/F_2.
- *Aligned pair of instr.*: merged; always executed.
- *Unaligned inst.:* executed conditionally.

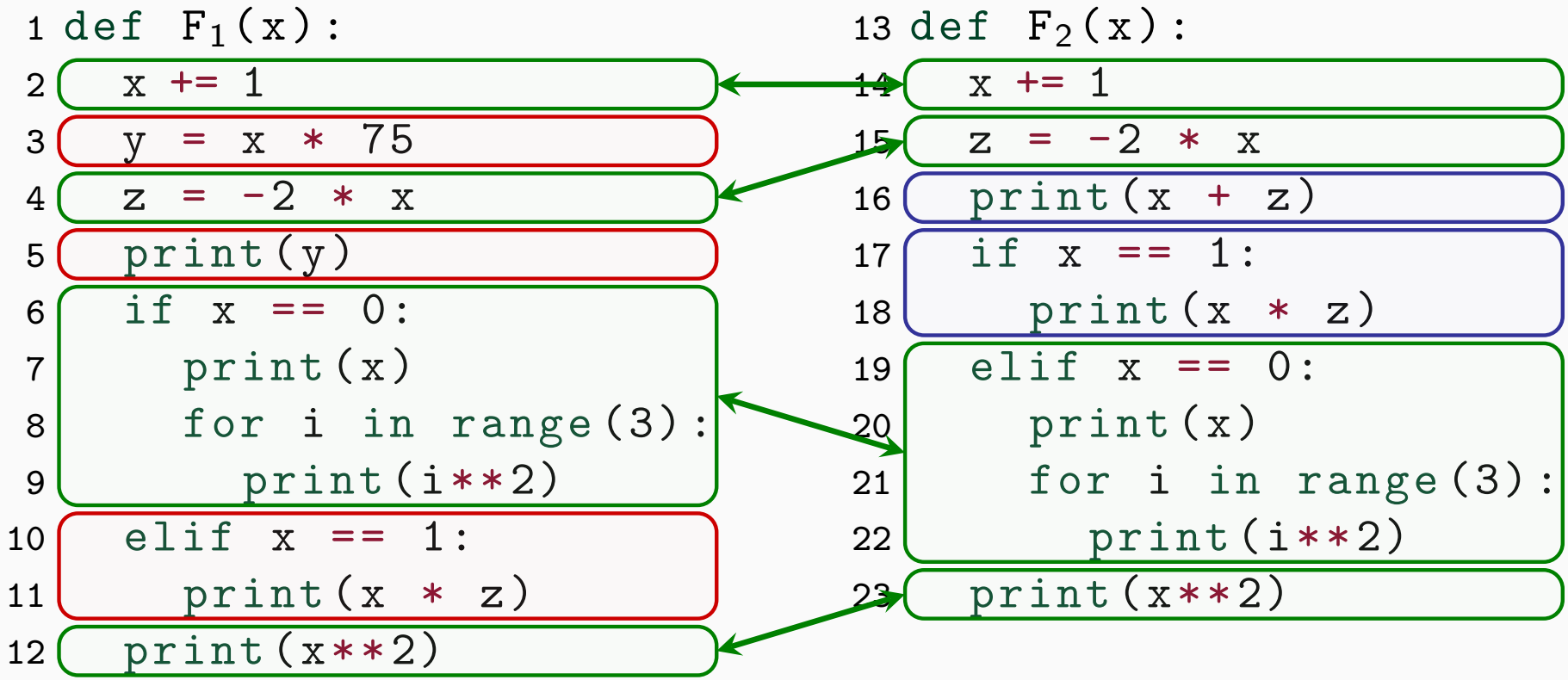
More aligned instructions $\Rightarrow$ smaller $F_\star$.

Function merging via sequence alignment (ex.)

```
1  def F1(x):
2    x += 1
3    y = x * 75
4    z = -2 * x
5    print(y)
6    if x == 0:
7      print(x)
8      for i in range(3):
9        print(i**2)
10   elif x == 1:
11     print(x * z)
12   print(x**2)
```

```
13 def F2(x):
14   x += 1
15   z = -2 * x
16   print(x + z)
17   if x == 1:
18     print(x * z)
19   elif x == 0:
20     print(x)
21     for i in range(3):
22       print(i**2)
23   print(x**2)
```

Function merging via sequence alignment (ex.)



Function merging via sequence alignment (ex.)

```
1  def F*(x, b):
2    x += 1
3    if b == 1:
4      y = x * 75
5    z = -2 * x
6    if b == 1:
7      print(y)
8    else:
9      print(x + z)
10   if b == 2 and x == 1:
11     print(x * z)
12   elif x == 0:
13     print(x)
14     for i in range(3):
15       print(i**2)
16   elif b == 1 and x == 1:
17     print(x * z)
18   print(x**2)
```

Function merging via sequence alignment

SOTA approach: **sequence alignment** [CGO'19, PLDI'20].

More aligned instructions $\Rightarrow$ smaller $F_\star$.

Function merging via sequence alignment

SOTA approach: **sequence alignment** [CGO'19, PLDI'20].

More aligned instructions $\Rightarrow$ smaller $F_\star$.

Sequence Alignment

Input: Two strings S_1 and S_2.

Output: Optimal sequence alignment of S_1 and S_2.

Function merging via sequence alignment

SOTA approach: **sequence alignment** [CGO'19, PLDI'20].

More aligned instructions $\Rightarrow$ smaller $F_\star$.

Sequence Alignment

Input: Two strings S_1 and S_2.

Output: Optimal sequence alignment of S_1 and S_2.

$\texttt{FunctionMerging}(F_1, F_2) \Rightarrow \texttt{SequenceAlignment}(F_1, F_2)$.

Function merging via sequence alignment

SOTA approach: **sequence alignment** [CGO'19, PLDI'20].

More aligned instructions $\Rightarrow$ smaller $F_\star$.

Sequence Alignment

Input: Two strings S_1 and S_2.

Output: Optimal sequence alignment of S_1 and S_2.

$\texttt{FunctionMerging}(F_1, F_2) \Rightarrow \texttt{SequenceAlignment}(F_1, F_2)$.

Textbook dynamic programming for Sequence Alignment
$\Longrightarrow \mathcal{O}(|F_1| \cdot |F_2|)$-time algorithm.

Sequence alignment + branch reordering

$\texttt{FunctionMerging}(F_1, F_2) \Rightarrow \texttt{SequenceAlignment}(F_1, F_2)$.

Drawback: a function is taken as a **fixed string**.

Sequence alignment + branch reordering

$\texttt{FunctionMerging}(F_1, F_2) \Rightarrow \texttt{SequenceAlignment}(F_1, F_2)$.

Drawback: a function is taken as a **fixed string**.

We can obtain different strings by **reordering branches**.

Sequence alignment + branch reordering

$\texttt{FunctionMerging}(F_1, F_2) \Rightarrow \texttt{SequenceAlignment}(F_1, F_2)$.

Drawback: a function is taken as a **fixed string**.

We can obtain different strings by **reordering branches**.

Reorder(F): all strings obtained by reordering branches in F.

Sequence alignment + branch reordering

$$\texttt{FunctionMerging}(F_1, F_2) \Rightarrow \texttt{SequenceAlignment}(F_1, F_2).$$

Drawback: a function is taken as a **fixed string**.

We can obtain different strings by **reordering branches**.

Reorder(F): all strings obtained by reordering branches in F.

$$\texttt{FunctionMerging}(F_1, F_2)$$

$$\Downarrow$$

$$\max_{\substack{F_1' \in \mathit{Reorder}(F_1) \\ F_2' \in \mathit{Reorder}(F_2)}} \texttt{SequenceAlignment}(F_1', F_2')$$

Sequence alignment + branch reordering (ex.)

```
1  def F1(x):
2    x += 1
3    y = x * 75
4    z = -2 * x
5    print(y)
6    if x == 0:
7      print(x)
8      for i in range(3):
9        print(i**2)
10   elif x == 1:
11     print(x * z)
12   print(x**2)
```

```
13 def F2(x):
14   x += 1
15   z = -2 * x
16   print(x + z)
17   if x == 1:
18     print(x * z)
19   elif x == 0:
20     print(x)
21     for i in range(3):
22       print(i**2)
23   print(x**2)
```

Sequence alignment + branch reordering (ex.)

```
1  def F1(x):
2      x += 1
3      y = x * 75
4      z = -2 * x
5      print(y)
6      if x == 0:
7          print(x)
8          for i in range(3):
9              print(i**2)
10     elif x == 1:
11         print(x * z)
12     print(x**2)
```

```
13 def F2(x):
14     x += 1
15     z = -2 * x
16     print(x + z)
17     if x == 1:
18         print(x * z)
19     elif x == 0:
20         print(x)
21         for i in range(3):
22             print(i**2)
23     print(x**2)
```

Sequence alignment + branch reordering (ex.)

```
1  def F'_1(x):
2    x += 1
3    y = x * 75
4    z = -2 * x
5    print(y)
6    if x == 1:
7      print(x * z)
8    elif x == 0:
9      print(x)
10     for i in range(3):
11       print(i**2)
12   print(x**2)
```

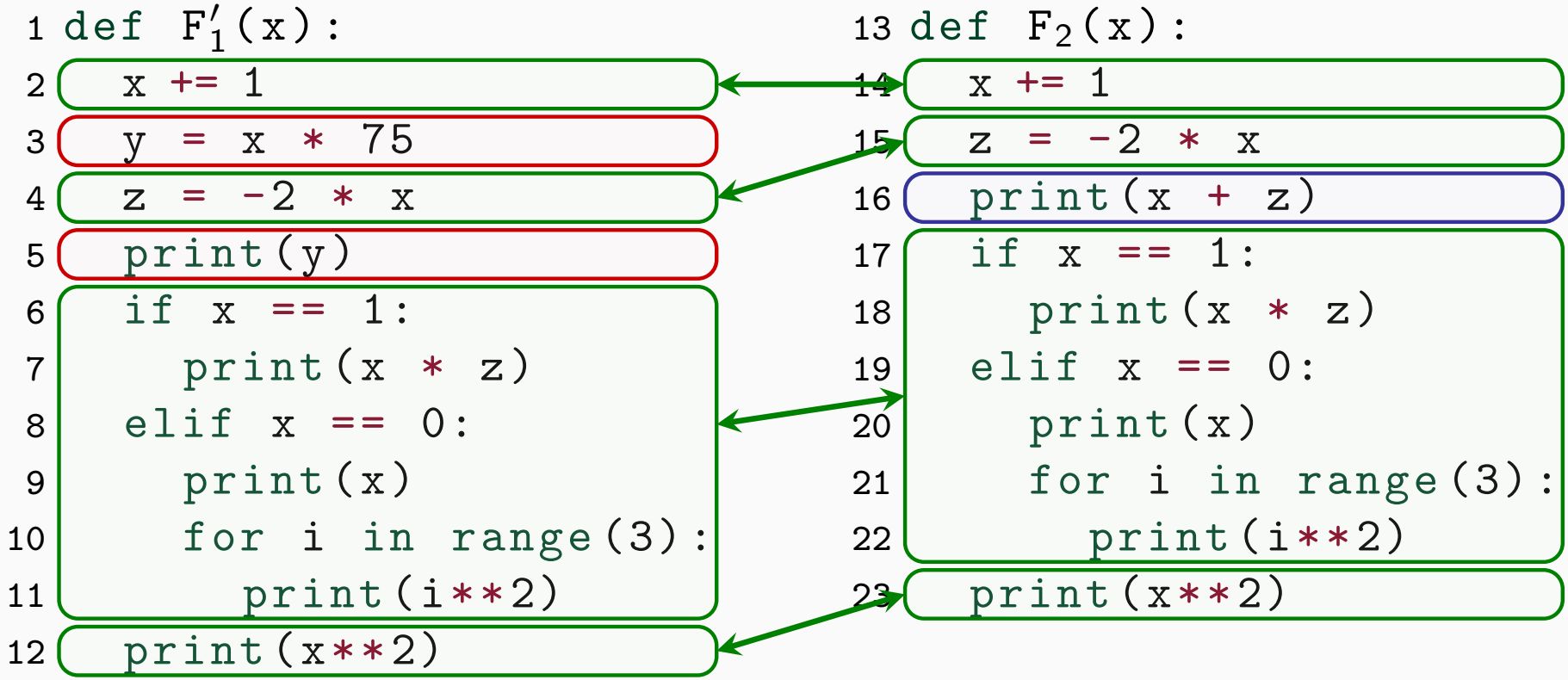
```
13 def F_2(x):
14   x += 1
15   z = -2 * x
16   print(x + z)
17   if x == 1:
18     print(x * z)
19   elif x == 0:
20     print(x)
21     for i in range(3):
22       print(i**2)
23   print(x**2)
```

Sequence alignment + branch reordering (ex.)

```
1  def F'_1(x):
2    x += 1
3    y = x * 75
4    z = -2 * x
5    print(y)
6    if x == 1:
7      print(x * z)
8    elif x == 0:
9      print(x)
10     for i in range(3):
11       print(i**2)
12   print(x**2)
```

```
13 def F_2(x):
14   x += 1
15   z = -2 * x
16   print(x + z)
17   if x == 1:
18     print(x * z)
19   elif x == 0:
20     print(x)
21     for i in range(3):
22       print(i**2)
23   print(x**2)
```

Sequence alignment + branch reordering (ex.)



Sequence alignment + branch reordering (ex.)

```
1  def F'⋆(x, b):
2    x += 1
3    if b == 1:
4      y = x * 75
5    z = -2 * x
6    if b == 1:
7      print(y)
8    else:
9      print(x + z)
10   if x == 1:
11     print(x * z)
12   elif x == 0:
13     print(x)
14     for i in range(3):
15       print(i**2)
16   print(x**2)
```

```
17 def F1(x):
18   F'⋆(x, 1)

19 def F2(x):
20   F'⋆(x, 2)
```

Sequence alignment + branch reordering (ex.)

```
1  def F'⋆(x, b):
2    x += 1
3    if b == 1:
4      y = x * 75
5    z = -2 * x
6    if b == 1:
7      print(y)
8    else:
9      print(x + z)
10   if x == 1:
11     print(x * z)
12   elif x == 0:
13     print(x)
14     for i in range(3):
15       print(i**2)
16   print(x**2)
```

```
17 def F1(x):
18   F'⋆(x, 1)

19 def F2(x):
20   F'⋆(x, 2)
```

Sequence alignment + branch reordering (ex.)

23 lines $\rightarrow$ 22 lines $\rightarrow$ 20 lines!

Problem statement

Function Merging with Branch Reordering

Input: Two functions F_1 and F_2.

Output: Compute

$$\max_{\substack{F_1' \in Reorder(F_1) \\ F_2' \in Reorder(F_2)}} \texttt{SequenceAlignment}(F_1', F_2')$$

Problem statement

Function Merging with Branch Reordering

Input: Two functions F_1 and F_2.

Output: Compute

$$\max_{\substack{F_1' \in Reorder(F_1) \\ F_2' \in Reorder(F_2)}} \texttt{SequenceAlignment}(F_1', F_2')$$

We study **algorithms and complexity** of this problem.

Complexity

Theorem (Hardness)

Function Merging with Branch Reordering is NP-hard.

Complexity

Theorem (Hardness)

Function Merging with Branch Reordering is NP-hard.

Workaround: parameterized algorithms!

Complexity

Theorem (Hardness)

Function Merging with Branch Reordering is NP-hard.

Workaround: parameterized algorithms!

Identify some **parameters** of the input such that:

(1) On inputs of interest, these parameters are **restricted**.

(2) Restricting the parameters allows for fast algorithms.

Complexity

Theorem (Hardness)

Function Merging with Branch Reordering is NP-hard.

Workaround: parameterized algorithms!

Identify some **parameters** of the input such that:

(1) On inputs of interest, these parameters are **restricted**.

(2) Restricting the parameters allows for fast algorithms.

Satisfying (1) and (2) $\Rightarrow$ profit!

Parameters: branching factor and nesting depth

Branching factor: max. # branches at any branching point.

Nesting depth: max. depth of nested branches.

Parameters: branching factor and nesting depth

Branching factor: max. # branches at any branching point.

Nesting depth: max. depth of nested branches.

```
1  def F1(x):
2    x += 1
3    y = x * 75
4    z = -2 * x
5    print(y)
6    if x == 0:
7      print(x)
8      for i in range(3):
9        print(i**2)
10   elif x == 1:
11     print(x * z)
12   print(x**2)
```

Parameters: branching factor and nesting depth

Branching factor: max. # branches at any branching point.

Nesting depth: max. depth of nested branches.

```
1  def F1(x):
2    x += 1
3    y = x * 75
4    z = -2 * x
5    print(y)
6    if x == 0:
7      print(x)
8      for i in range(3):
9        print(i**2)
10   elif x == 1:
11     print(x * z)
12   print(x**2)
```

Branching factor of $F_1 = 2$.

Parameters: branching factor and nesting depth

Branching factor: max. # branches at any branching point.

Nesting depth: max. depth of nested branches.

```
1  def F1(x):
2      x += 1
3      y = x * 75
4      z = -2 * x
5      print(y)
6      if x == 0:
7          print(x)
8          for i in range(3):
9              print(i**2)
10     elif x == 1:
11         print(x * z)
12     print(x**2)
```

Branching factor of $F_1 = 2$.

Nesting depth of $F_1 = 2$.

Parameters: branching factor and nesting depth

Branching factor: max. # branches at any branching point.

Nesting depth: max. depth of nested branches.

Hypothesis: These parameters are typically small.

Two parameterized algorithms

Four parameters:

- branching factor of F_1/F_2 (b_1/b_2),
- nesting depth of F_1/F_2 (d_1/d_2).

Two parameterized algorithms

Four parameters:

- branching factor of F_1/F_2 (b_1/b_2),
- nesting depth of F_1/F_2 (d_1/d_2).

Theorem (Restricting b_1, b_2, d_1, d_2)

Function Merging with Branch Reordering can be solved in

$$2^{\mathcal{O}(b_1 d_1 + b_2 d_2)} \cdot |F_1| \cdot |F_2| \text{ time.}$$

Two parameterized algorithms

Four parameters:

- branching factor of F_1/F_2 (b_1/b_2),
- nesting depth of F_1/F_2 (d_1/d_2).

Theorem (Restricting b_1, b_2, d_1, d_2)

Function Merging with Branch Reordering can be solved in

$$2^{\mathcal{O}(b_1 d_1 + b_2 d_2)} \cdot |F_1| \cdot |F_2| \text{ time.}$$

Theorem (Restricting b_1, b_2, d_1)

Function Merging with Branch Reordering can be solved in

$$2^{\mathcal{O}(b_1 d_1 + b_2)} \cdot (|F_1| \cdot |F_2|)^4 \text{ time.}$$

Two parameterized algorithms

Theorem (Restricting b_1, b_2, d_1, d_2)

Function Merging with Branch Reordering can be solved in

$$2^{\mathcal{O}(b_1 d_1 + b_2 d_2)} \cdot |F_1| \cdot |F_2| \text{ time.}$$

Theorem (Restricting b_1, b_2, d_1)

Function Merging with Branch Reordering can be solved in

$$2^{\mathcal{O}(b_1 d_1 + b_2)} \cdot (|F_1| \cdot |F_2|)^4 \text{ time.}$$

Other restrictions?

Two parameterized algorithms

Theorem (Restricting b_1, b_2, d_1, d_2)

Function Merging with Branch Reordering can be solved in

$$2^{\mathcal{O}(b_1 d_1 + b_2 d_2)} \cdot |F_1| \cdot |F_2| \text{ time.}$$

Theorem (Restricting b_1, b_2, d_1)

Function Merging with Branch Reordering can be solved in

$$2^{\mathcal{O}(b_1 d_1 + b_2)} \cdot (|F_1| \cdot |F_2|)^4 \text{ time.}$$

Other restrictions?

For most of them, the hardness theorem applies.

Two parameterized algorithms

Theorem (Restricting b_1, b_2, d_1, d_2)

Function Merging with Branch Reordering can be solved in

$$2^{\mathcal{O}(b_1 d_1 + b_2 d_2)} \cdot |F_1| \cdot |F_2| \text{ time.}$$

Theorem (Restricting b_1, b_2, d_1)

Function Merging with Branch Reordering can be solved in

$$2^{\mathcal{O}(b_1 d_1 + b_2)} \cdot (|F_1| \cdot |F_2|)^4 \text{ time.}$$

Other restrictions?

For most of them, the hardness theorem applies.

Open: what if b_1, b_2 are restricted, but d_1, d_2 are unbounded?

Overview of the first algorithm

Function Merging with Branch Reordering

Input: Two functions F_1 and F_2.

Output: Compute

$$\max_{\substack{F_1' \in Reorder(F_1) \\ F_2' \in Reorder(F_2)}} \texttt{SequenceAlignment}(F_1', F_2')$$

Overview of the first algorithm

Function Merging with Branch Reordering

Input: Two functions F_1 and F_2.

Output: Compute

$$\max_{\substack{F_1' \in Reorder(F_1) \\ F_2' \in Reorder(F_2)}} \texttt{SequenceAlignment}(F_1', F_2')$$

Bottleneck: $Reorder(F)$ has size $\Omega\left(2^{|F|}\right)$.

Overview of the first algorithm

Function Merging with Branch Reordering

Input: Two functions F_1 and F_2.

Output: Compute

$$\max_{\substack{F_1' \in Reorder(F_1) \\ F_2' \in Reorder(F_2)}} \texttt{SequenceAlignment}(F_1', F_2')$$

Bottleneck: $Reorder(F)$ has size $\Omega\left(2^{|F|}\right)$.

dp algorithm for $\texttt{SequenceAlignment}(S_1, S_2)$

$\approx$

incremental, **simultaneous** scanning of S_1 and S_2.

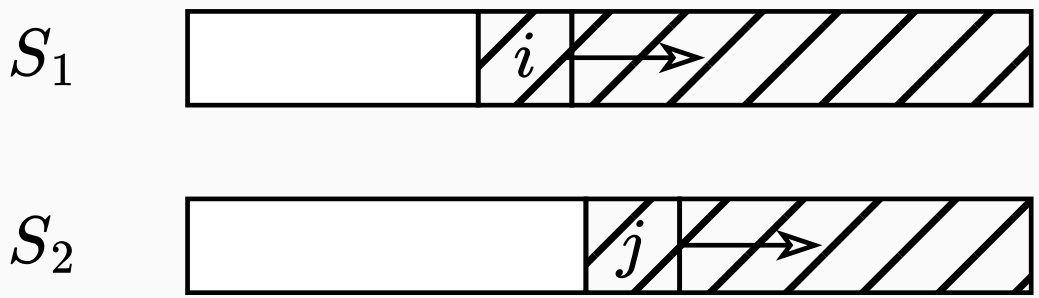
Overview of the first algorithm

dp algorithm for `SequenceAlignment`(S_1, S_2)

$\approx$

incremental, **simultaneous** scanning of S_1 and S_2.

$$dp[i, j] = \texttt{SequenceAlignment}(S_1[i \ldots], S_2[j \ldots])$$



Overview of the first algorithm

dp algorithm for `SequenceAlignment`(S_1, S_2)

$\approx$

incremental, **simultaneous** scanning of S_1 and S_2.

$\Downarrow$

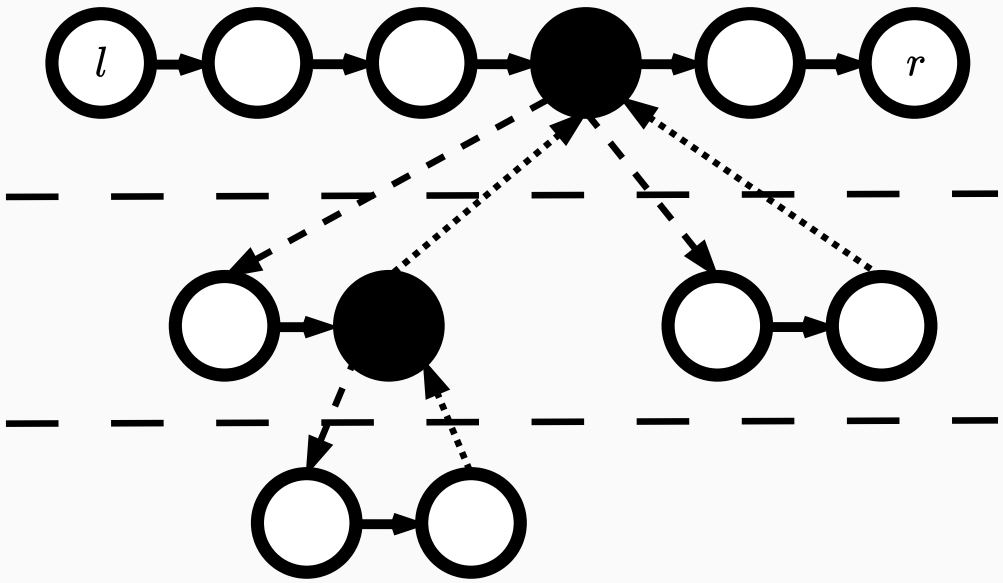
incremental, **simultaneous** exploration of $\mathit{Reorder}(F_1)$ and $\mathit{Reorder}(F_2)$; compute sequence alignment **on the fly**.

Programs to branching graphs

```
1  def F(x):
2    x += 1
3    y = x * 75
4    z = -2 * x
5    if x == 0:
6      print(x)
7      for i in range(3):
8        print(i)
9        print(i**2)
10   elif x == 1:
11     print(z)
12     print(x * z)
13   print(x**2)
14   print(z**3)
```

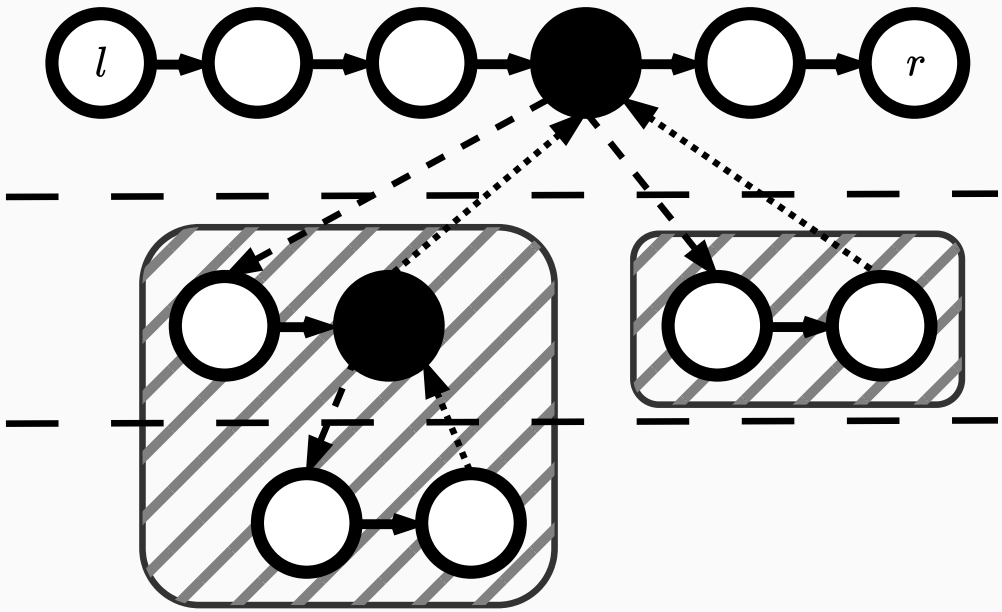
Programs to branching graphs

```
1  def F(x):
2      x += 1
3      y = x * 75
4      z = -2 * x
5      if x == 0:
6          print(x)
7          for i in range(3):
8              print(i)
9              print(i**2)
10     elif x == 1:
11         print(z)
12         print(x * z)
13     print(x**2)
14     print(z**3)
```

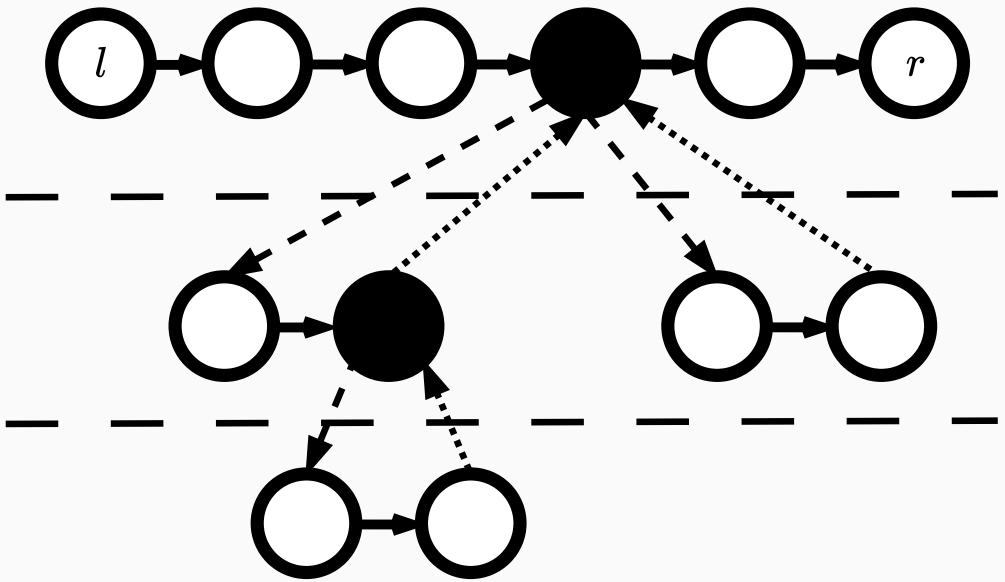


Programs to branching graphs

```
1  def F(x):
2      x += 1
3      y = x * 75
4      z = -2 * x
5      if x == 0:
6          print(x)
7          for i in range(3):
8              print(i)
9              print(i**2)
10     elif x == 1:
11         print(z)
12         print(x * z)
13     print(x**2)
14     print(z**3)
```



Exploring *Reorder*(*F*)



$\textsc{Explore}(\ell, \emptyset, \varepsilon)$.

procedure $\textsc{Explore}(u, X, S)$

if X contains all nodes: **print** S; **return.**

$X \leftarrow X \cup \{u\}$.

if u is ○: $\textsc{Explore}(\mathit{next}(u),\ X,\ S \cdot \mathit{instr}(u))$.

if u is ●: **forall** $v \in \mathit{branches}(u) \setminus X$: $\textsc{Explore}(v, X, S)$.

Exploring *Reorder*(F)

EXPLORE($\ell, \emptyset, \varepsilon$).

procedure EXPLORE(u, X, S)

  if X contains all nodes: **print** S; **return.**

  $X \leftarrow X \cup \{u\}$.

  if u is ○: EXPLORE($\mathit{next}(u),\ X,\ S \cdot \mathit{instr}(u)$).

  if u is ●: **forall** $v \in \mathit{branches}(u) \setminus X$: EXPLORE($v, X, S$).

Lemma (Compactness)

The number of distinct (u, X) *encountered in* EXPLORE *is at most*

$$2^{\mathcal{O}(bd)} \cdot |F|.$$

Overview of the first algorithm

$$\textsc{Explore}(u, X, \dots)$$

$$\Downarrow$$

$$\textsc{FunctionMerging}(u_1, X_1, u_2, X_2)$$

Overview of the first algorithm

$$\textsc{Explore}(u, X, \dots)$$

$$\Downarrow$$

$$\textsc{FunctionMerging}(u_1, X_1, u_2, X_2)$$

\# dynamic programming states: $2^{\mathcal{O}(b_1 d_1 + b_2 d_2)} \cdot |F_1| \cdot |F_2|$.

Work per state: $\text{poly}(b_1, b_2, d_1, d_2) \cdot (|F_1| + |F_2|)$.

Overview of the first algorithm

$$\textsc{Explore}(u, X, \dots)$$
$$\Downarrow$$
$$\textsc{FunctionMerging}(u_1, X_1, u_2, X_2)$$

\# dynamic programming states: $2^{\mathcal{O}(b_1 d_1 + b_2 d_2)} \cdot |F_1| \cdot |F_2|$.

Work per state: $\text{poly}(b_1, b_2, d_1, d_2) \cdot (|F_1| + |F_2|)$.

Optimized $X_1, X_2 \Rightarrow \text{poly}(b_1, b_2, d_1, d_2)$ work per state.

Overview of the first algorithm

$$\textsc{Explore}(u, X, \dots)$$
$$\Downarrow$$
$$\textsc{FunctionMerging}(u_1, X_1, u_2, X_2)$$

\# dynamic programming states: $2^{\mathcal{O}(b_1 d_1 + b_2 d_2)} \cdot |F_1| \cdot |F_2|$.

Work per state: $\text{poly}(b_1, b_2, d_1, d_2) \cdot (|F_1| + |F_2|)$.

Optimized $X_1, X_2 \Rightarrow \text{poly}(b_1, b_2, d_1, d_2)$ work per state.

Theorem (Restricting b_1, b_2, d_1, d_2)

Function Merging with Branch Reordering can be solved in

$$2^{\mathcal{O}(b_1 d_1 + b_2 d_2)} \cdot |F_1| \cdot |F_2| \text{ time.}$$

Overview of the second algorithm

Theorem (Restricting b_1, b_2, d_1)

Function Merging with Branch Reordering can be solved in

$$2^{\mathcal{O}(b_1 d_1 + b_2)} \cdot (|F_1| \cdot |F_2|)^4 \text{ time.}$$

View first algorithm as

weighted reachability on a finite graph

of size $2^{\mathcal{O}(b_1 d_1 + b_2 d_2)} \cdot |F_1| \cdot |F_2|$.

Overview of the second algorithm

Theorem (Restricting b_1, b_2, d_1)

Function Merging with Branch Reordering can be solved in

$$2^{\mathcal{O}(b_1 d_1 + b_2)} \cdot (|F_1| \cdot |F_2|)^4 \text{ time.}$$

View first algorithm as

weighted reachability on a finite graph

of size $2^{\mathcal{O}(b_1 d_1 + b_2 d_2)} \cdot |F_1| \cdot |F_2|$.

Encode X_2 as a **stack** structure $\Rightarrow$

reachability on a weighted pushdown system.

Overview of the second algorithm

Theorem (Restricting b_1, b_2, d_1)

Function Merging with Branch Reordering can be solved in

$$2^{\mathcal{O}(b_1 d_1 + b_2)} \cdot (|F_1| \cdot |F_2|)^4 \text{ time.}$$

View first algorithm as

weighted reachability on a finite graph

of size $2^{\mathcal{O}(b_1 d_1 + b_2 d_2)} \cdot |F_1| \cdot |F_2|$.

Encode X_2 as a **stack** structure $\Rightarrow$

reachability on a weighted pushdown system.

Use classical reachability algorithm [SAS'03].