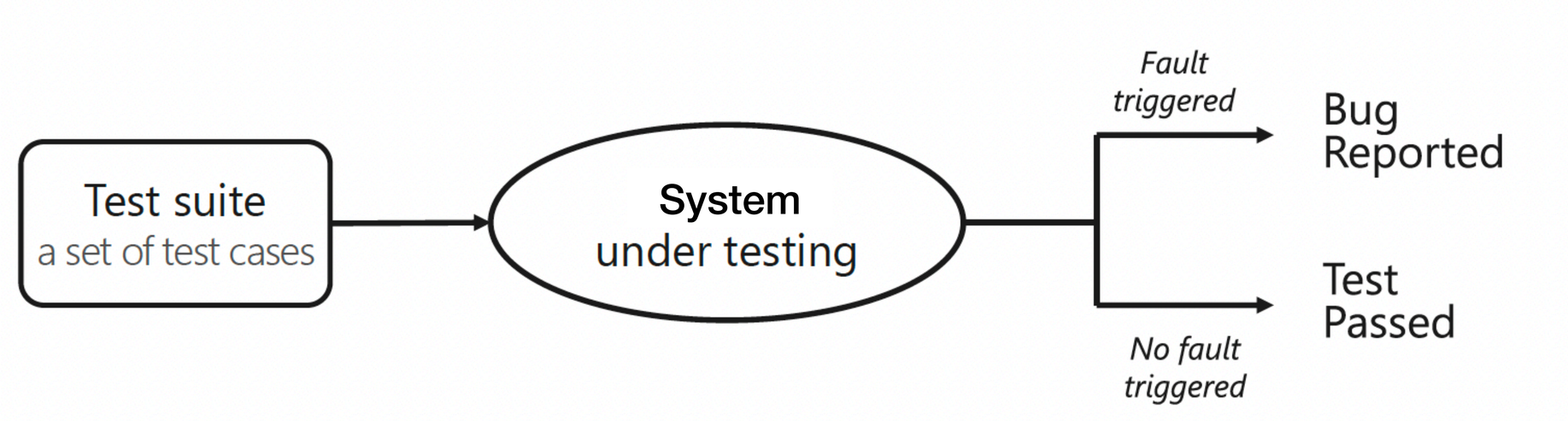
CAmpactor: A Novel and Effective Local Search Algorithm for Optimizing Pairwise Covering Arrays

Qiyuan Zhao, Chuan Luo, Shaowei Cai, Wei Wu, Jinkun Lin, Hongyu Zhang, Chunming Hu

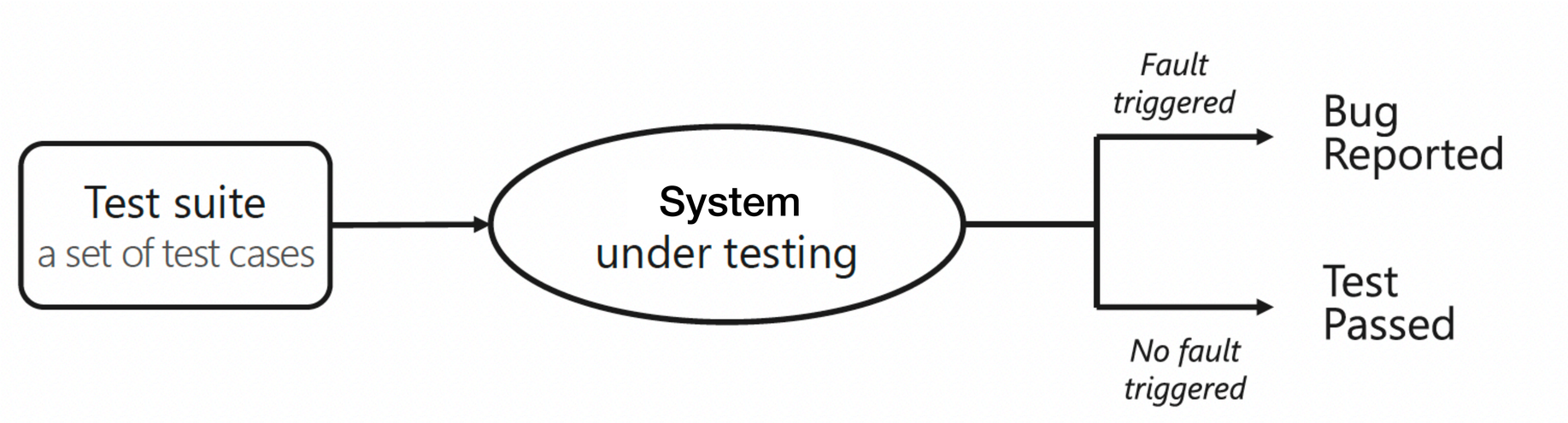
Image credit: https://en.wikipedia.org/wiki/Compactor

For ESEC/FSE 2023

Combinatorial Interaction Testing

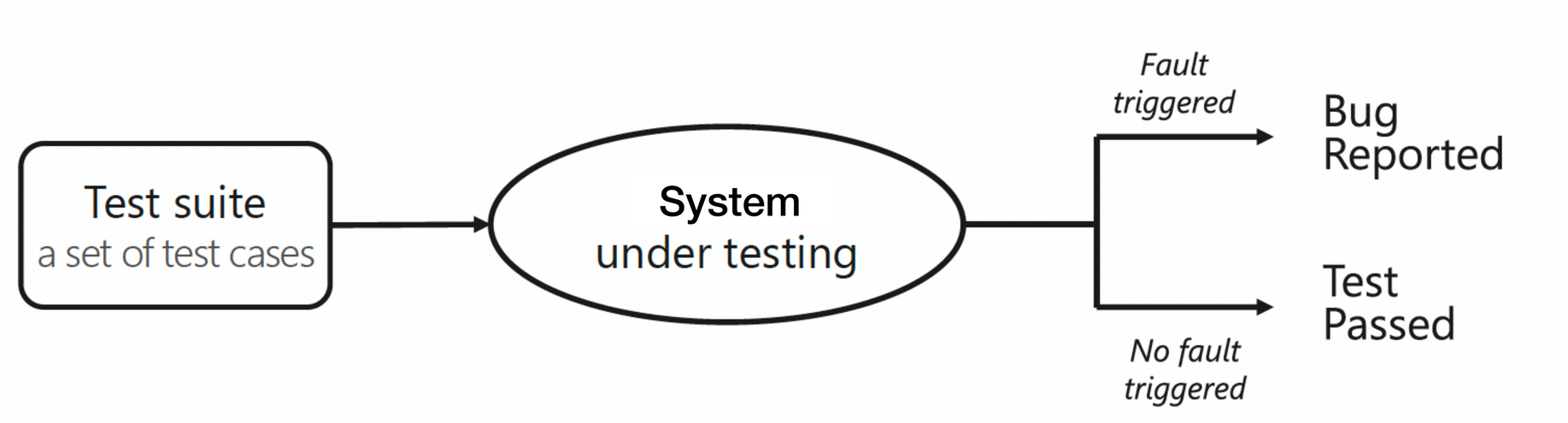


Combinatorial Interaction Testing



- System under testing: modeled as set of multi-valued **options**

Combinatorial Interaction Testing



- System under testing: modeled as set of multi-valued **options**
- **Test case***:* configuration of options

Combinatorial Interaction Testing (cont'd)

- **Combinatorial interaction testing**: detecting faults triggered by combinations of t option-value pairs

Combinatorial Interaction Testing (cont'd)

Table: System under Testing (SUT)

Operating System	Browser	Payment Method
Windows	Internet Explorer	Alipay
Linux	Chrome	Paypal
Mac OS	Microsoft Edge	Wechat
Android	Safari	Visa

- **Combinatorial interaction testing**: detecting faults triggered by combinations of t option-value pairs

Combinatorial Interaction Testing (cont'd)

option

Table: System under Testing (SUT)

Operating System	Browser	Payment Method
Windows	Internet Explorer	Alipay
Linux	Chrome	Paypal
Mac OS	Microsoft Edge	Wechat
Android	Safari	Visa

- **Combinatorial interaction testing**: detecting faults triggered by combinations of t option-value pairs

Combinatorial Interaction Testing (cont'd)

option

Table: System under Testing (SUT)

Operating System	Browser	Payment Method
Windows	Internet Explorer	Alipay
Linux	Chrome	Paypal
Mac OS	Microsoft Edge	Wechat
Android	Safari	Visa

value of option

- **Combinatorial interaction testing**: detecting faults triggered by combinations of t option-value pairs

Combinatorial Interaction Testing (cont'd)

option

Table: System under Testing (SUT)

Operating System	Browser	Payment Method
Windows	Internet Explorer	Alipay
Linux	Chrome	Paypal
Mac OS	Microsoft Edge	Wechat
Android	Safari	Visa

value of option

- **Combinatorial interaction testing**: detecting faults triggered by combinations of t option-value pairs (or, t**-wise tuples**)

Combinatorial Interaction Testing (cont'd)

option

Table: System under Testing (SUT)

Operating System	Browser	Payment Method
Windows	Internet Explorer	Alipay
Linux	Chrome	Paypal
Mac OS	Microsoft Edge	Wechat
Android	Safari	Visa

value of option

- **Combinatorial interaction testing**: detecting faults triggered by combinations of t option-value pairs (or, t**-wise tuples**)
  - **Pairwise testing**: t fixed to 2

Combinatorial Interaction Testing (cont'd)

option

Table: System under Testing (SUT)

Operating System	Browser	Payment Method
Windows	Internet Explorer	Alipay
Linux	Chrome	Paypal
Mac OS	Microsoft Edge	Wechat
Android	Safari	Visa

value of option

- **Combinatorial interaction testing**: detecting faults triggered by combinations of t option-value pairs (or, t**-wise tuples**)
  - **Pairwise testing**: t fixed to 2
  - 2-wise tuple: referred to as **pairwise tuple**

Combinatorial Interaction Testing (cont'd)

Table: System under Testing (SUT)

Operating System	Browser	Payment Method
Windows	Internet Explorer	Alipay
Linux	Chrome	Paypal
Mac OS	Microsoft Edge	Wechat
Android	Safari	Visa

- **Combinatorial interaction testing**: detecting faults triggered by combinations of t option-value pairs (or, t**-wise tuples**)
  - **Pairwise testing**: t fixed to 2
  - 2-wise tuple: referred to as **pairwise tuple**
    - E.g., τ = {Operating System=Android, Payment Method=Wechat}

Combinatorial Interaction Testing (cont’d)

- **Test case***:* complete assignment of options

Combinatorial Interaction Testing (cont'd)

- **Test case***:* complete assignment of options
  - E.g., tc = {Operating System=Android, Browser=Chrome, Payment Method=Wechat}

Combinatorial Interaction Testing (cont'd)

- **Test case***:* complete assignment of options
  - E.g., tc = {Operating System=Android, Browser=Chrome, Payment Method=Wechat}
- Pairwise tuple τ is **covered** by test case $tc \overset{\text{def}}{=} \tau \subseteq tc$

Combinatorial Interaction Testing (cont'd)

- **Test case***:* complete assignment of options
  - E.g., tc = {Operating System=Android, Browser=Chrome, Payment Method=Wechat}
- Pairwise tuple τ is **covered** by test case $tc \overset{\text{def}}{=} \tau \subseteq tc$
  - E.g., {Operating System=Android, Payment Method=Wechat} is covered by {Operating System=Android, Browser=Chrome, Payment Method=Wechat}

Combinatorial Interaction Testing (cont’d)

- **Constraints** can exist over options and values

Combinatorial Interaction Testing (cont'd)

- **Constraints** can exist over options and values
  - E.g., "if Operating System=Linux, then Browser≠Safari"

Combinatorial Interaction Testing (cont'd)

- **Constraints** can exist over options and values
  - E.g., "if Operating System=Linux, then Browser≠Safari"
- A pairwise tuple/test case is **valid** iff it respects all constraints

Combinatorial Interaction Testing (cont'd)

- **Constraints** can exist over options and values
  - E.g., "if Operating System=Linux, then Browser≠Safari"
- A pairwise tuple/test case is **valid** iff it respects all constraints

- **Pairwise Covering Array (PCA)**:

Combinatorial Interaction Testing (cont'd)

- **Constraints** can exist over options and values
  - E.g., "if Operating System=Linux, then Browser≠Safari"
- A pairwise tuple/test case is **valid** iff it respects all constraints

- **Pairwise Covering Array (PCA)**: a set T of valid test cases, such that every valid pairwise tuple is covered by a test case in T

Combinatorial Interaction Testing (cont'd)

- **Constraints** can exist over options and values
  - E.g., "if Operating System=Linux, then Browser≠Safari"
- A pairwise tuple/test case is **valid** iff it respects all constraints

- **Pairwise Covering Array (PCA)**: a set T of valid test cases, such that every valid pairwise tuple is covered by a test case in T
  - The **size** of a PCA is the number of test cases it contains

Combinatorial Interaction Testing (cont'd)

- **Constraints** can exist over options and values
  - E.g., "if Operating System=Linux, then Browser≠Safari"
- A pairwise tuple/test case is **valid** iff it respects all constraints

- **Pairwise Covering Array (PCA)**: a set T of valid test cases, such that every valid pairwise tuple is covered by a test case in T
  - The **size** of a PCA is the number of test cases it contains
  - Any fault triggered by 2 options can be detected with a PCA

PCA Initialization and Optimization

- PCA with smaller size $\Longrightarrow$ still 100% pairwise coverage, less testing cost

PCA Initialization and Optimization

- PCA with smaller size $\Longrightarrow$ still 100% pairwise coverage, less testing cost
- The two-phase approach to building a small PCA:

PCA Initialization and Optimization

- PCA with smaller size $\Longrightarrow$ still 100% pairwise coverage, less testing cost
- The two-phase approach to building a small PCA:
  - **Initialization phase**: constructing PCA quickly

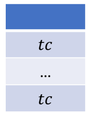
PCA Initialization and Optimization

- PCA with smaller size $\Longrightarrow$ still 100% pairwise coverage, less testing cost
- The two-phase approach to building a small PCA:
  - **Initialization phase**: constructing PCA quickly
  - **Optimization phase**: reducing the size of PCA (e.g., by **local search**)

PCA Optimization By Local Search

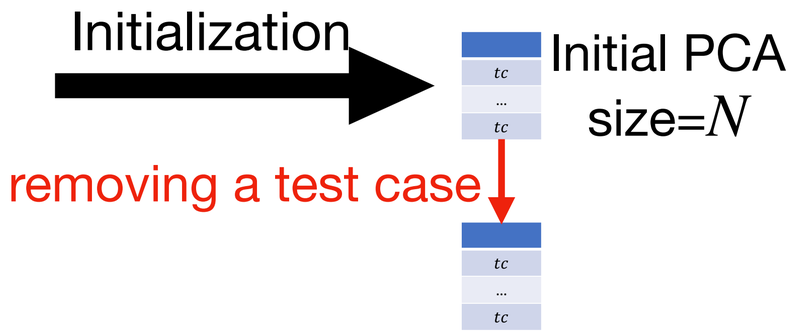
PCA Optimization By Local Search

Initialization

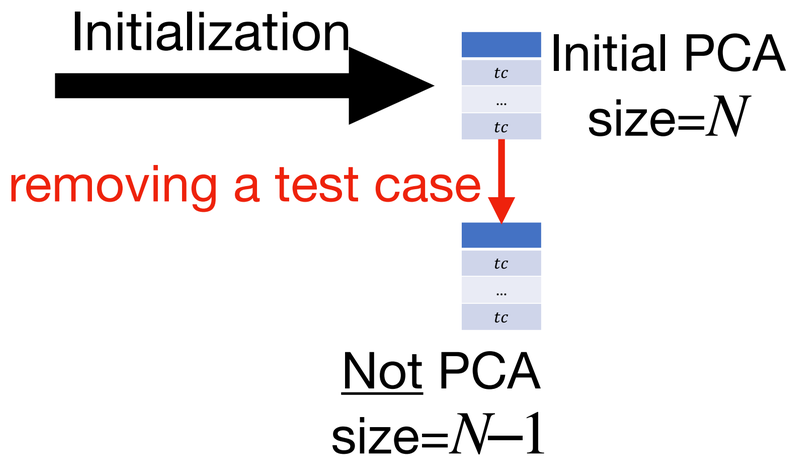


Initial PCA
size=N

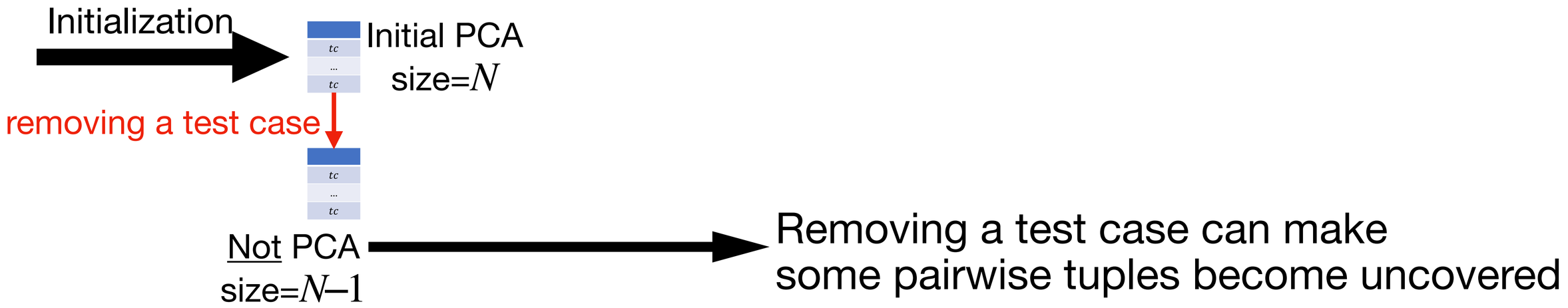
PCA Optimization By Local Search



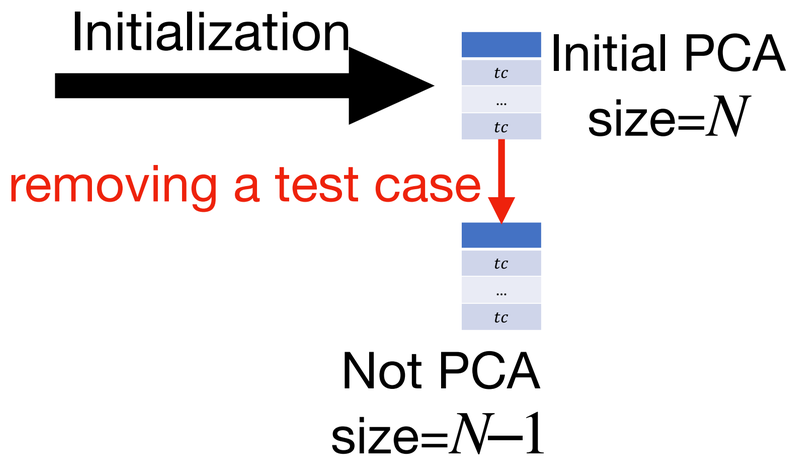
PCA Optimization By Local Search



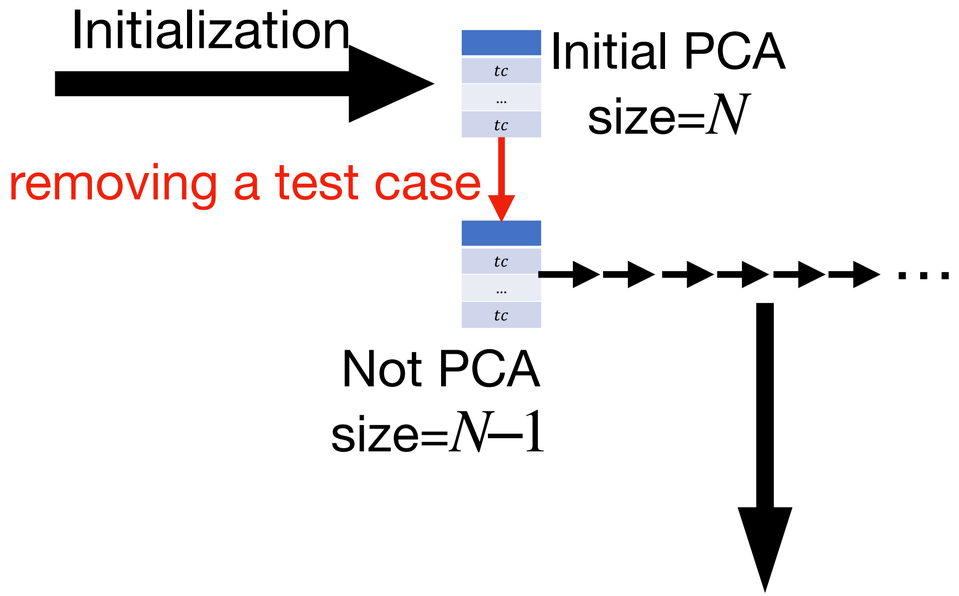
PCA Optimization By Local Search



PCA Optimization By Local Search

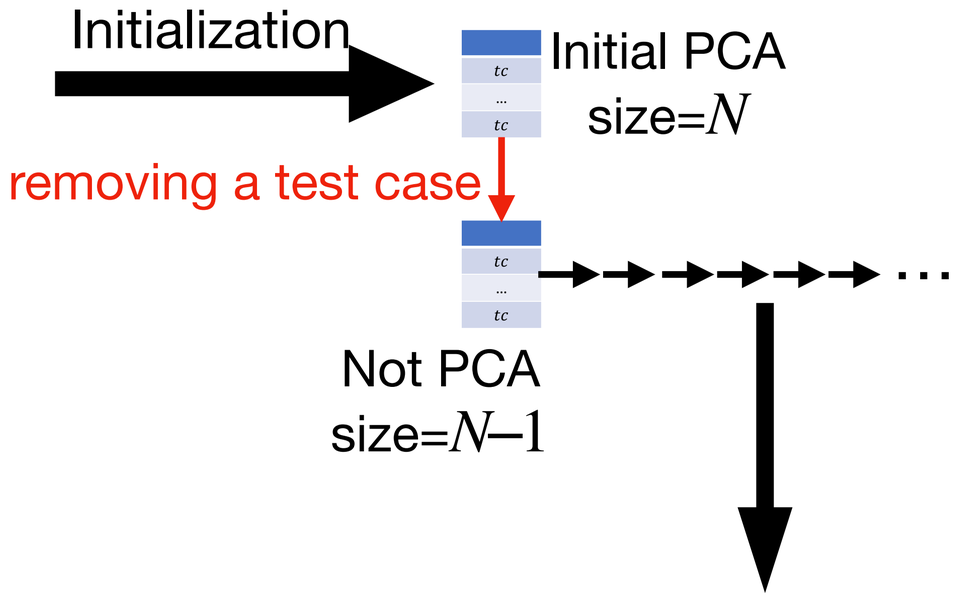


PCA Optimization By Local Search



Local search steps: modifying existing test cases in the set to make them cover the “lost” valid pairwise tuples

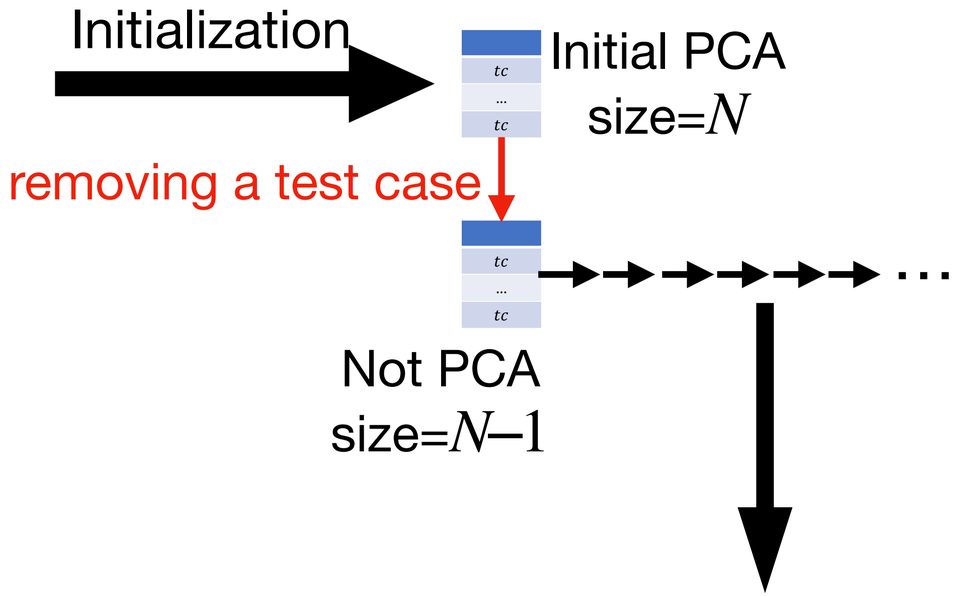
PCA Optimization By Local Search



Local search steps: modifying existing test cases in the set to make them cover the “lost” valid pairwise tuples

- Typically, only one test case will be modified in each step

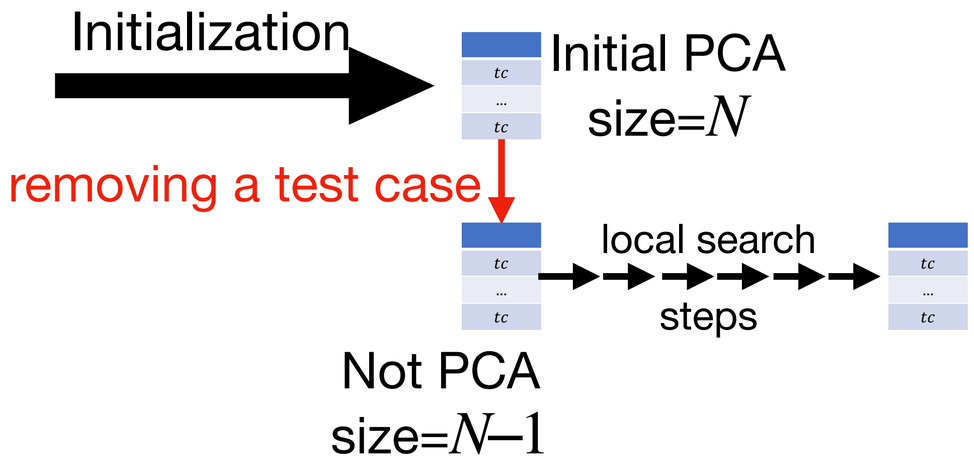
PCA Optimization By Local Search



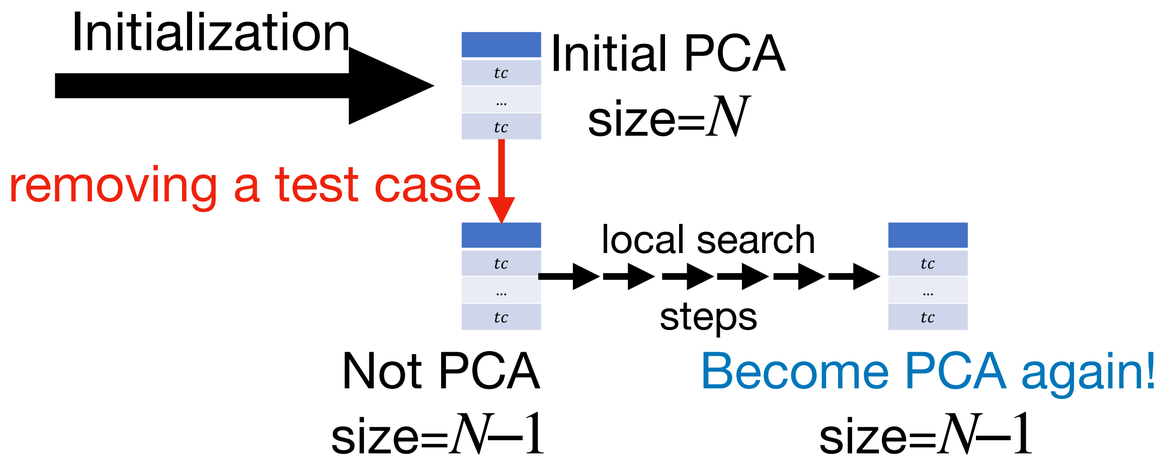
Local search steps: modifying existing test cases in the set to make them cover the “lost” valid pairwise tuples

- Typically, only one test case will be modified in each step
- The modified test case must be still valid

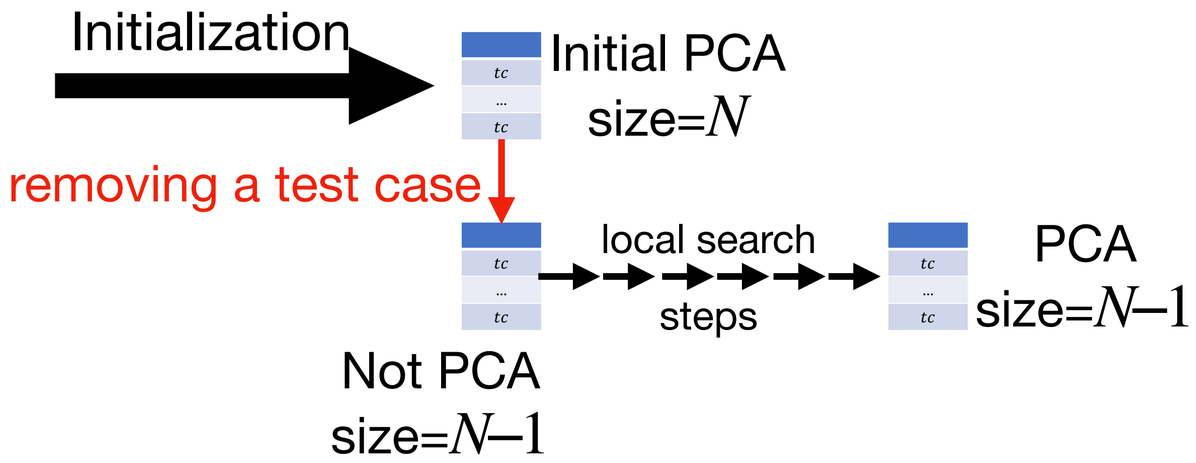
PCA Optimization By Local Search



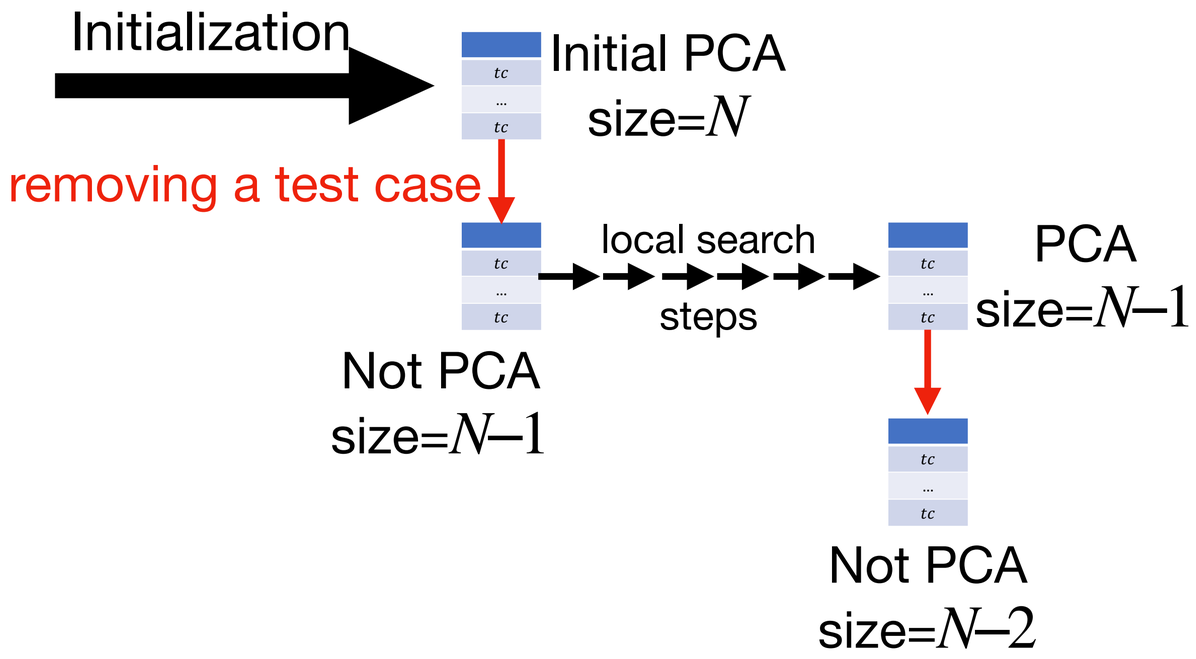
PCA Optimization By Local Search



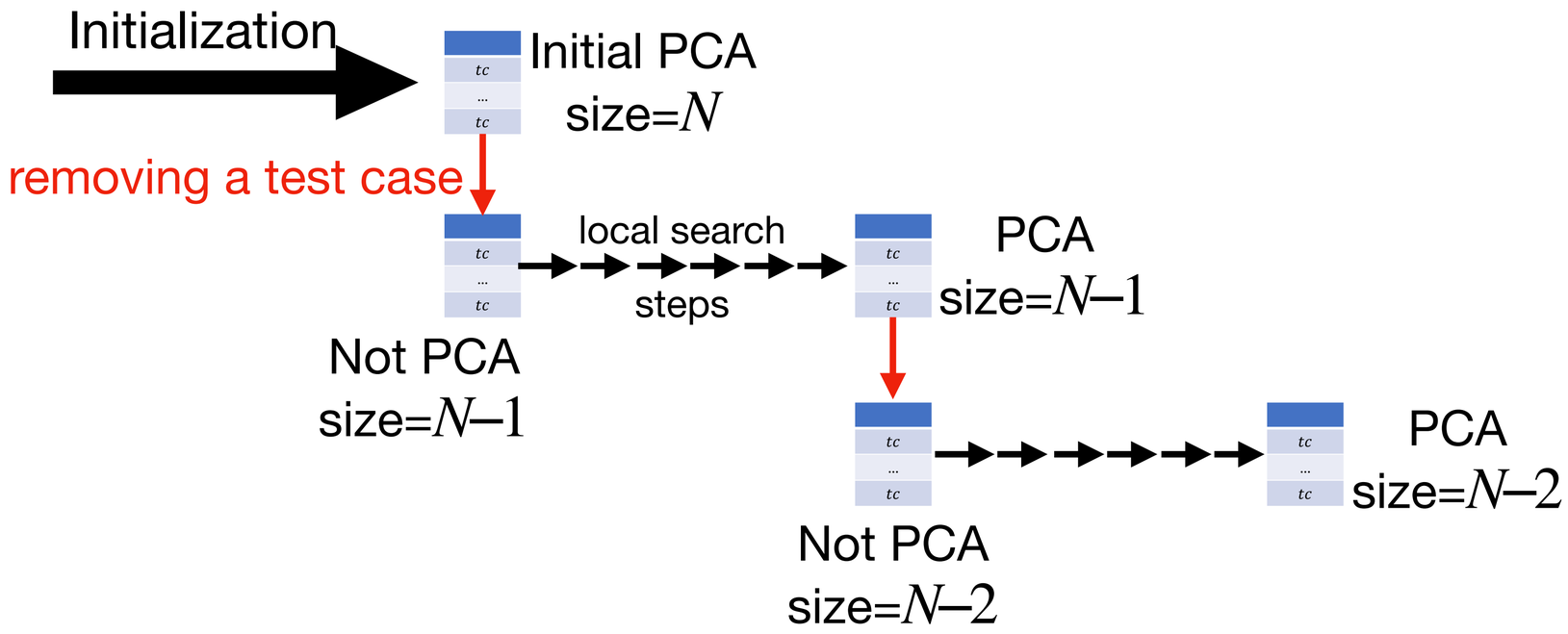
PCA Optimization By Local Search



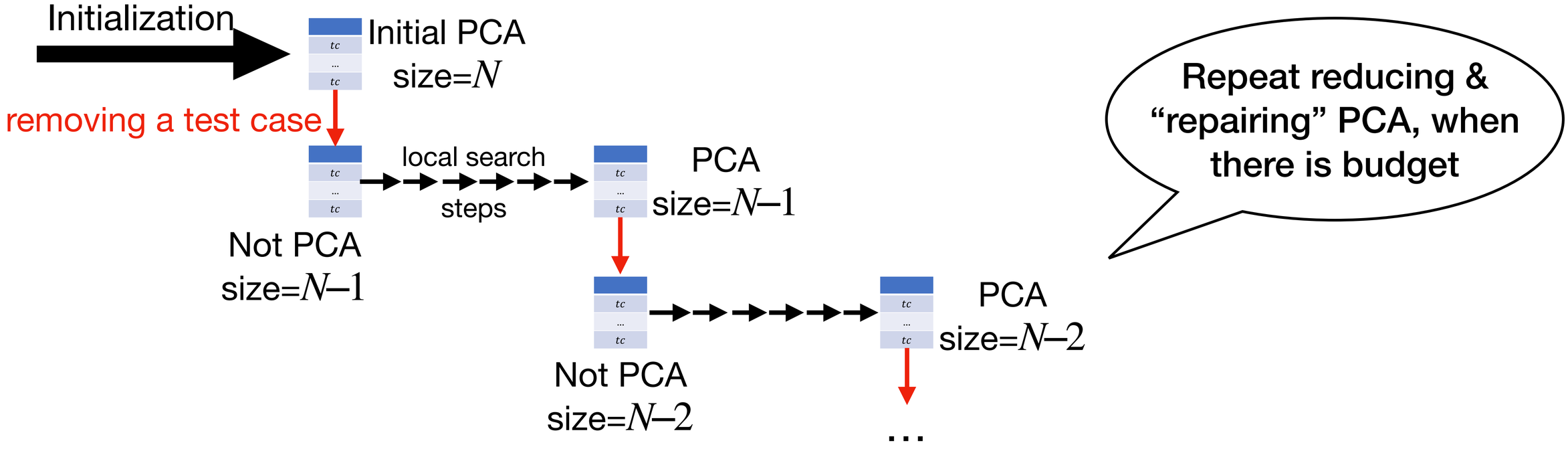
PCA Optimization By Local Search



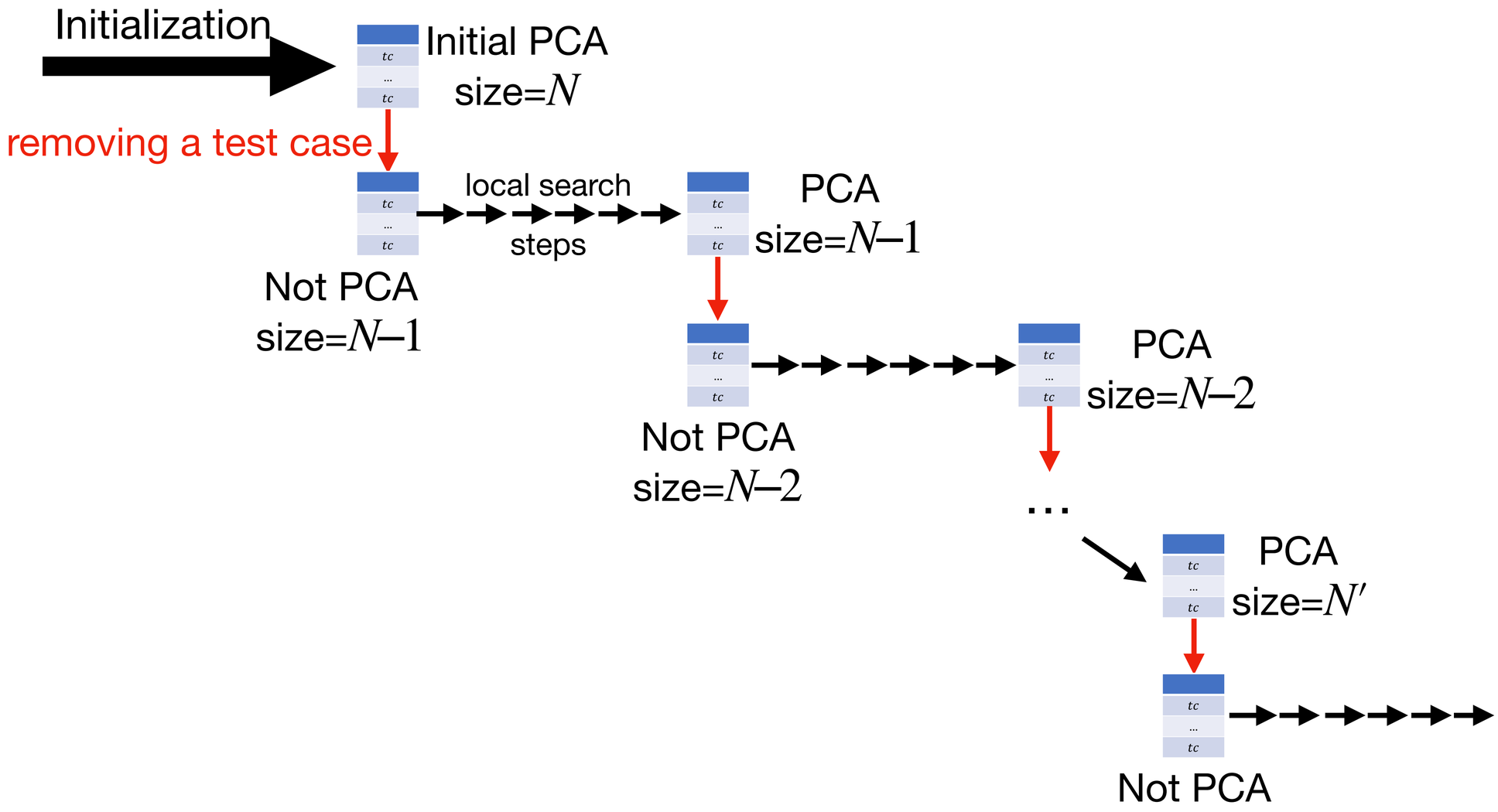
PCA Optimization By Local Search



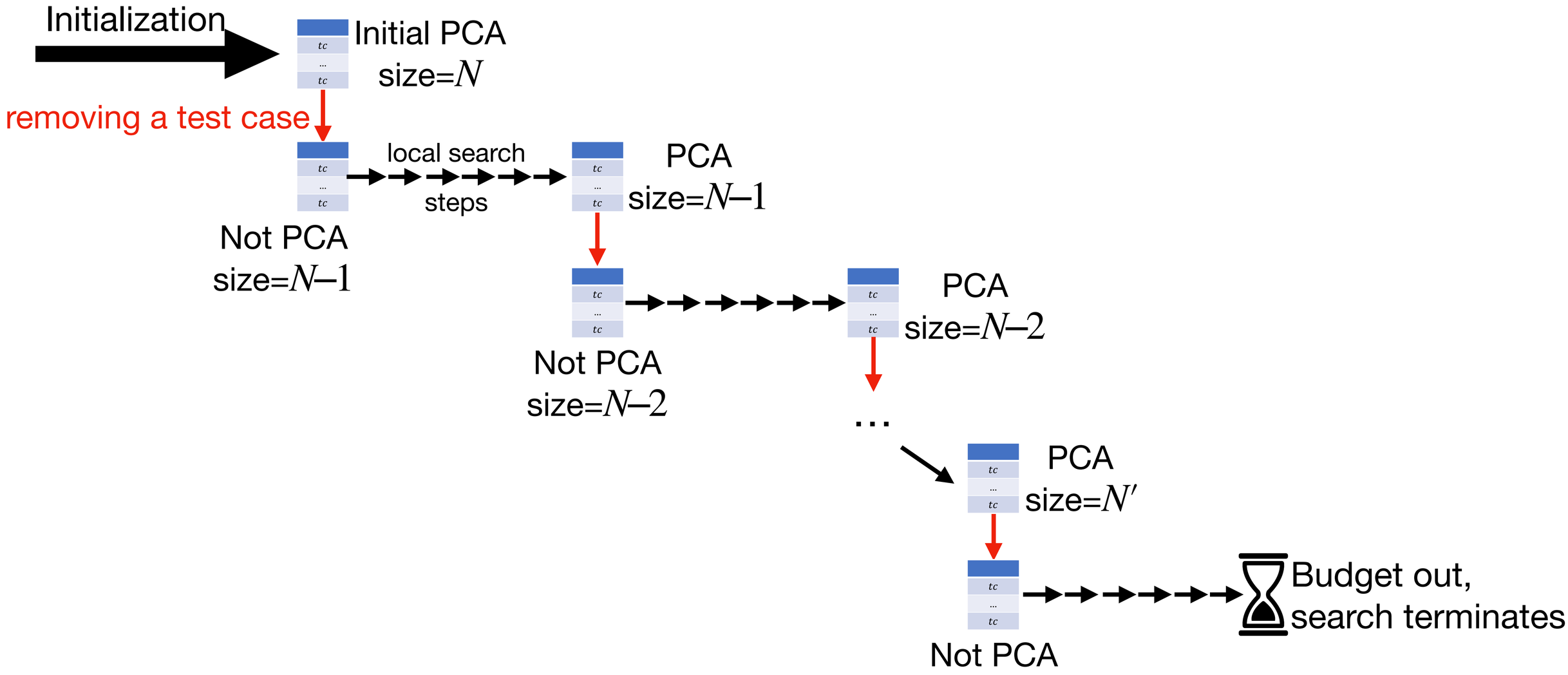
PCA Optimization By Local Search



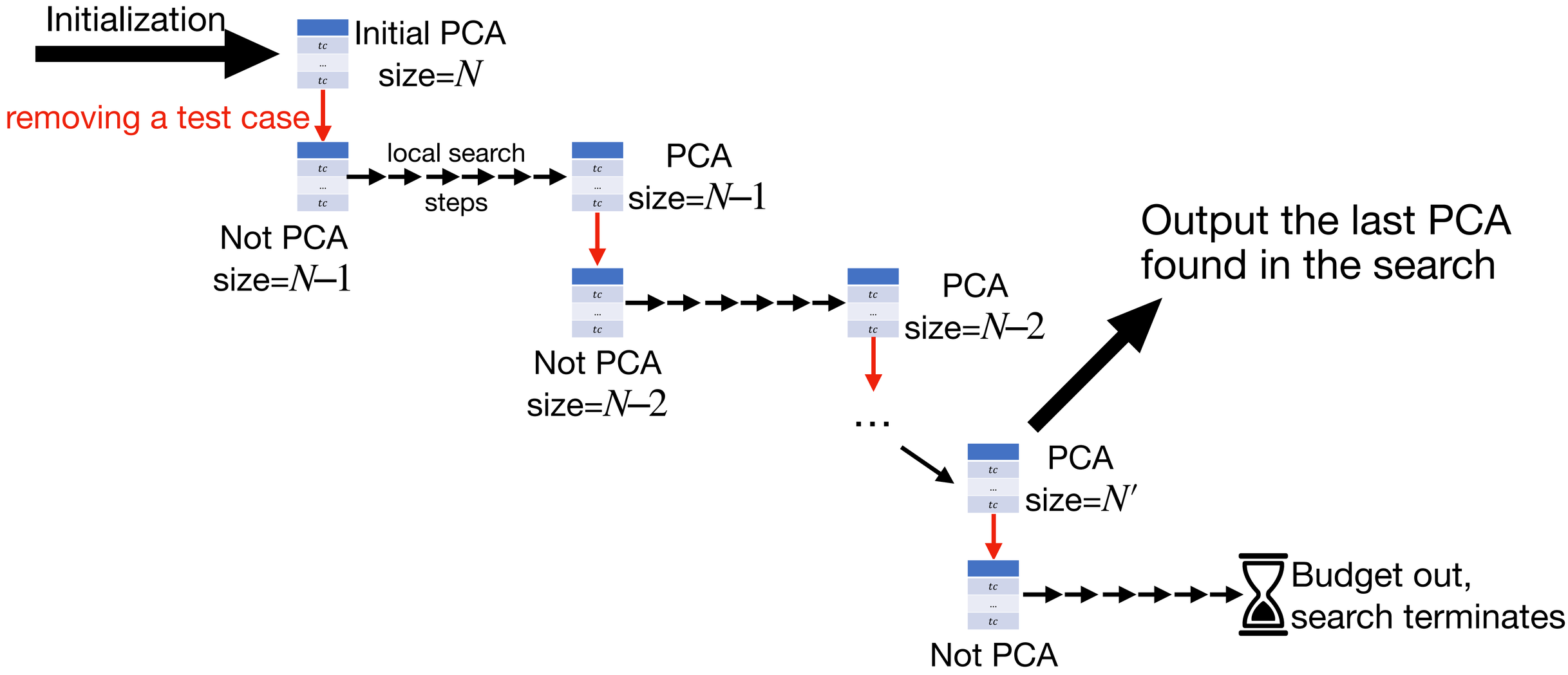
PCA Optimization By Local Search



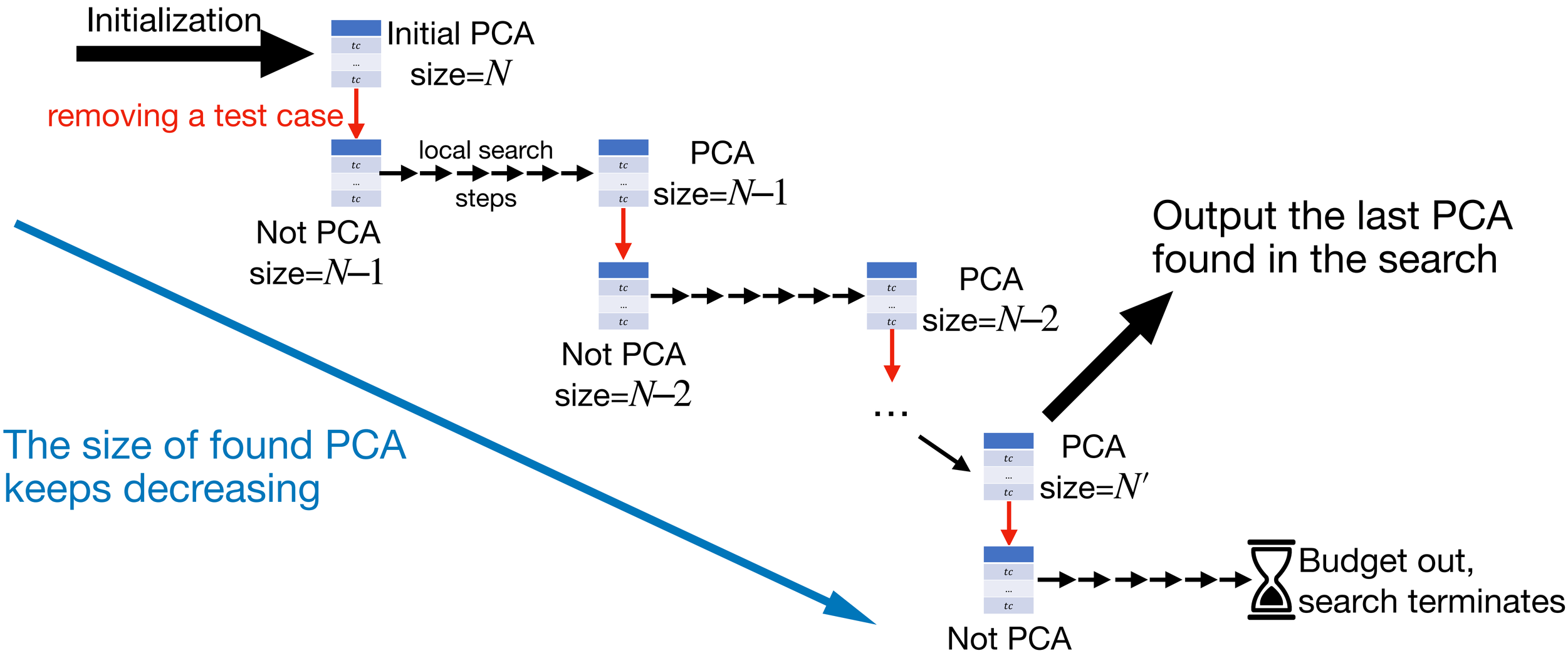
PCA Optimization By Local Search



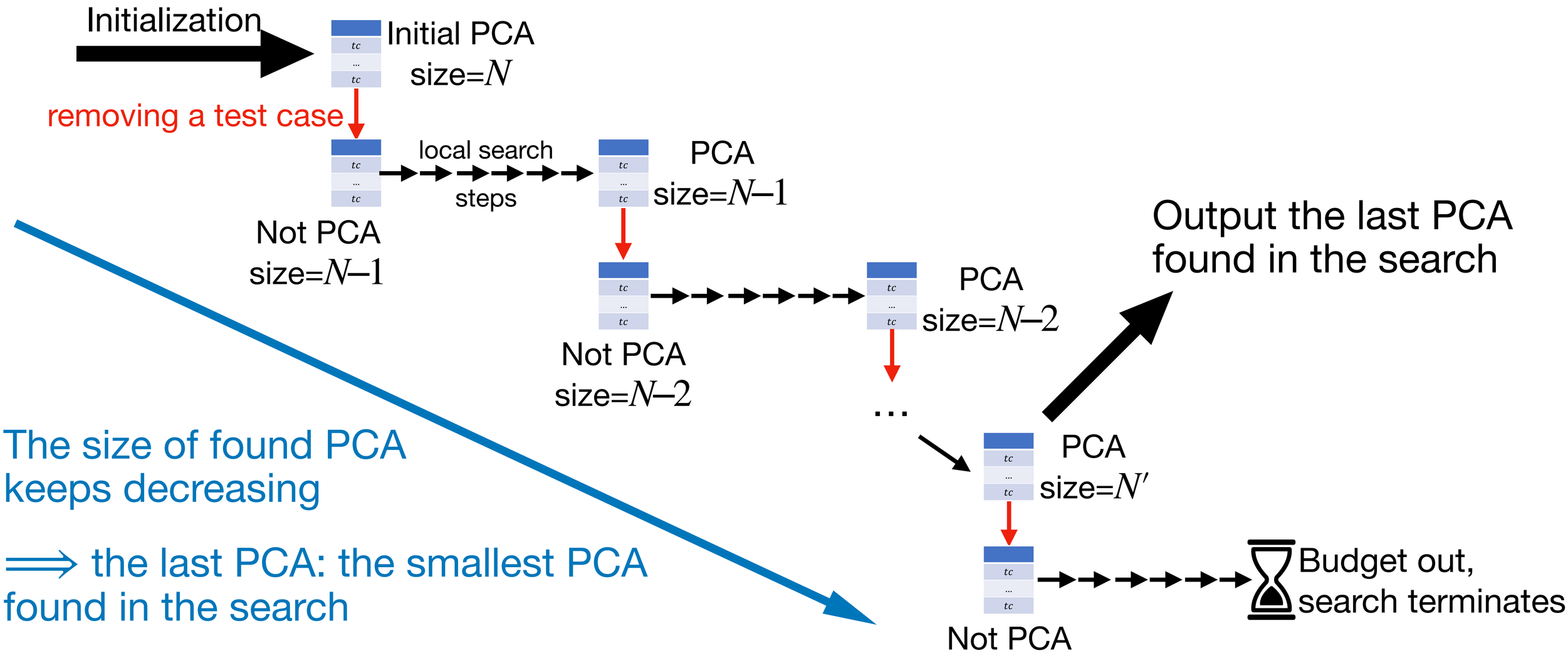
PCA Optimization By Local Search



PCA Optimization By Local Search



PCA Optimization By Local Search



Core Problem: Scalability

- PCA optimization can be **hard!**

Core Problem: Scalability

- PCA optimization can be **hard!**
- Existing local search PCA optimization algorithms cannot handle PCAs of large systems (e.g., systems with $\geq 10^3$ options and many constraints)

Core Problem: Scalability

- PCA optimization can be **hard!**
- Existing local search PCA optimization algorithms cannot handle PCAs of large systems (e.g., systems with $\geq 10^3$ options and many constraints)
  - Output PCAs are relatively large $\Longrightarrow$ ineffective

Core Problem: Scalability

- PCA optimization can be **hard!**
- Existing local search PCA optimization algorithms cannot handle PCAs of large systems (e.g., systems with $\geq 10^3$ options and many constraints)
  - Output PCAs are relatively large $\Longrightarrow$ ineffective

- *CAmpactor*: an effective local search algorithm for PCA optimization (or, “compacting” PCA)

Core Problem: Scalability

- PCA optimization can be **hard!**
- Existing local search PCA optimization algorithms cannot handle PCAs of large systems (e.g., systems with $\geq 10^3$ options and many constraints)
  - Output PCAs are relatively large $\Longrightarrow$ ineffective

- *CAmpactor*: an effective local search algorithm for PCA optimization (or, "compacting" PCA)
  - Equipped with special techniques to overcome the scalability problem

Challenge #1: Hindering Issue

Challenge #1: Hindering Issue

- In the existing work, the modification in a local search step is typically minor

Challenge #1: Hindering Issue

- In the existing work, the modification in a local search step is typically minor
  - E.g., changing a single option's value in a test case

Challenge #1: Hindering Issue

- In the existing work, the modification in a local search step is typically minor
  - E.g., changing a single option's value in a test case
- Some pairwise tuples can be very difficult to cover using only minor modifications

Challenge #1: Hindering Issue

- In the existing work, the modification in a local search step is typically minor
  - E.g., changing a single option's value in a test case
- Some pairwise tuples can be very difficult to cover using only minor modifications
  - Mainly due to complex constraints

Challenge #1: Hindering Issue

- In the existing work, the modification in a local search step is typically minor
  - E.g., changing a single option's value in a test case
- Some pairwise tuples can be very difficult to cover using only minor modifications
  - Mainly due to complex constraints
  - Making search stagnate $\Longrightarrow$ hindering PCA optimization from going further

Challenge #1: Hindering Issue (cont'd)

- In the existing work, the modification in a local search step is typically minor
- Solution: *CAmpactor* occasionally performs **forced patching**

Challenge #1: Hindering Issue (cont'd)

- In the existing work, the modification in a local search step is typically minor
- Solution: *CAmpactor* occasionally performs **forced patching**
  - Given an uncovered pairwise tuple τ, chooses a test case tc

Challenge #1: Hindering Issue (cont'd)

- In the existing work, the modification in a local search step is typically minor
- Solution: *CAmpactor* occasionally performs **forced patching**
  - Given an uncovered pairwise tuple τ, chooses a test case tc
  - Uses a specialized SAT solver to find a valid test case tc' which guarantees to cover τ, and is similar to tc

Challenge #1: Hindering Issue (cont'd)

- In the existing work, the modification in a local search step is typically minor
- Solution: *CAmpactor* occasionally performs **forced patching**
  - Given an uncovered pairwise tuple τ, chooses a test case tc
  - Uses a specialized SAT solver to find a valid test case tc' which guarantees to cover τ, and is similar to tc
  - Replaces tc with tc' (thereby τ will become covered)

Challenge #1: Hindering Issue (cont'd)

- In the existing work, the modification in a local search step is typically minor
- Solution: *CAmpactor* occasionally performs **forced patching**
  - Given an uncovered pairwise tuple τ, chooses a test case tc
  - Uses a specialized SAT solver to find a valid test case tc' which guarantees to cover τ, and is similar to tc
  - Replaces tc with tc' (thereby τ will become covered)
    - Whole test case replacement $\Longrightarrow$ major modification

Challenge #1: Hindering Issue (cont'd)

- In the existing work, the modification in a local search step is typically minor
- Solution: *CAmpactor* occasionally performs **forced patching**
  - Given an uncovered pairwise tuple τ, chooses a test case tc
  - Uses a specialized SAT solver to find a valid test case tc' which guarantees to cover τ, and is similar to tc
  - Replaces tc with tc' (thereby τ will become covered)
    - Whole test case replacement $\Longrightarrow$ major modification
    - Being similar $\Longrightarrow$ many pairwise tuples covered by tc are preserved after modification

Challenge #2: Cycling Issue

Challenge #2: Cycling Issue

- In the existing work, the **forbidden strategy** of local search is usually specified at the level of a single option's value

Challenge #2: Cycling Issue

- In the existing work, the **forbidden strategy** of local search is usually specified at the level of a single option's value
  - Not strong enough to prevent local search from stagnating in our case

Challenge #2: Cycling Issue

- In the existing work, the **forbidden strategy** of local search is usually specified at the level of a single option's value
  - Not strong enough to prevent local search from stagnating in our case
- Solution: *CAmpactor* has forbidden strategy at the level of test cases

Challenge #2: Cycling Issue

- In the existing work, the **forbidden strategy** of local search is usually specified at the level of a single option's value
  - Not strong enough to prevent local search from stagnating in our case
- Solution: *CAmpactor* has forbidden strategy at the level of test cases
  - Allowing optimization to go deeper

Evaluation

- Adopt a collection of benchmarking system models
  - With varying numbers of options and complexities of constraints
  - Modeled from real-world systems, adopted in many previous studies

Evaluation

- Adopt a collection of benchmarking system models
  - With varying numbers of options and complexities of constraints
  - Modeled from real-world systems, adopted in many previous studies
- State-of-the-art competitors: mainly *TCA*, *FastCA* and *AutoCCAG*

Evaluation

- Adopt a collection of benchmarking system models
  - With varying numbers of options and complexities of constraints
  - Modeled from real-world systems, adopted in many previous studies
- State-of-the-art competitors: mainly *TCA*, *FastCA* and *AutoCCAG*
  - Competitors are given the same PCA initialized by *SamplingCA*

Evaluation (cont’d)

- RQ1: comparison with SOTA competitors

Evaluation (cont'd)

- RQ1: comparison with SOTA competitors
- Result: *CAmpactor* produces roughly **<u>45%</u>** smaller PCAs than any other competitor in average $\Longrightarrow$ *CAmpactor* is effective

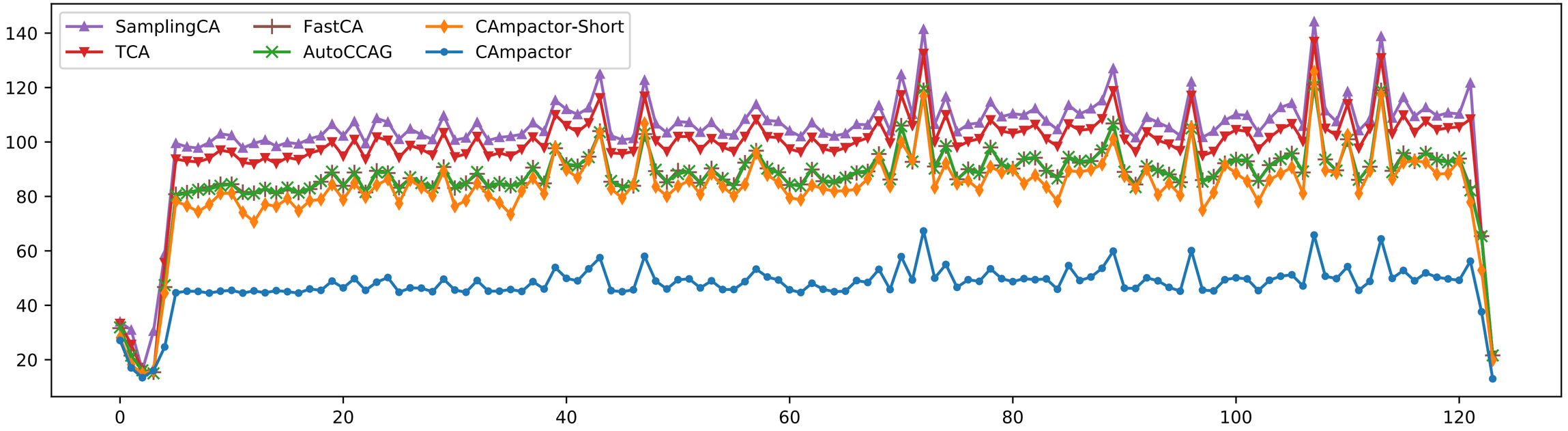


Figure 1 in the paper, X-axis: benchmark ID; Y-axis: PCA size in avg.

Evaluation (cont'd)

- RQ1: comparison with SOTA competitors
- Result: *CAmpactor* produces roughly **<u>45%</u>** smaller PCAs than any other competitor in average $\Longrightarrow$ *CAmpactor* is effective

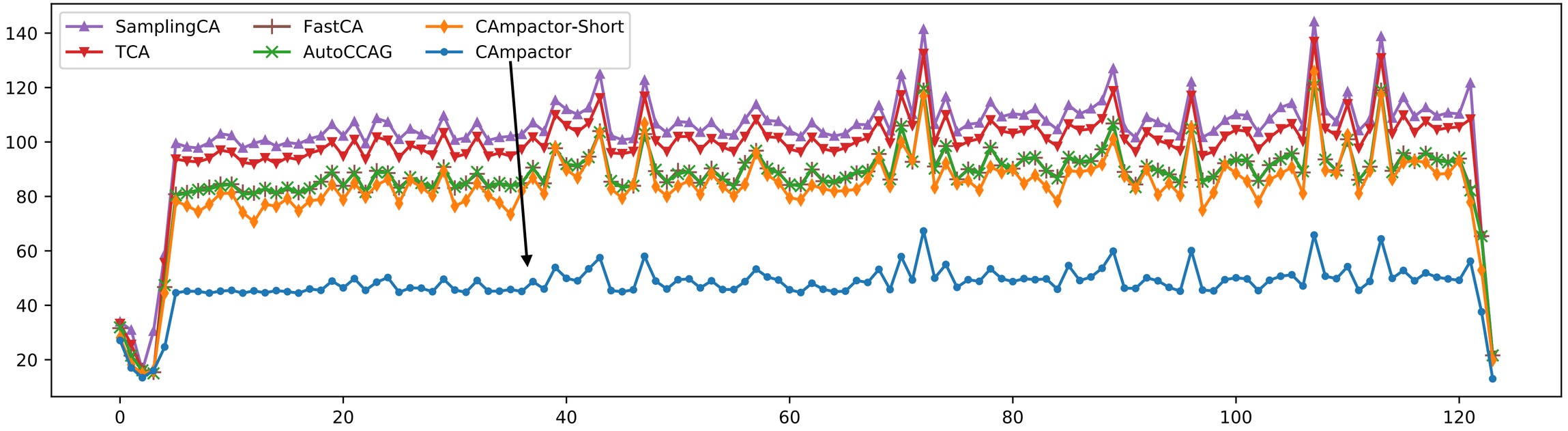


Figure 1 in the paper, X-axis: benchmark ID; Y-axis: PCA size in avg.

Evaluation (cont'd)

- RQ1: comparison with SOTA competitors
- Result: *CAmpactor* is also efficient

	CAmpactor	*Short**	*SamplingCA*	*AutoCCAG*	*FastCA*	*TCA*
avg. size	**47.4**	82.7	104.0	86.6	86.7	98.1
avg. time	284.6	52.7	42.1	377.9	357.2	52.7

*To save space, we use '*Short*' to denote '*CAmpactor-Short*'.

Table 2 in the paper; all the execution times includes that of *SamplingCA*
CAmpactor-Short uses the running time of *TCA* as cutoff time

Evaluation (cont'd)

- RQ1: comparison with SOTA competitors
- Result: *CAmpactor* is also efficient

	CAmpactor	*Short**	*SamplingCA*	*AutoCCAG*	*FastCA*	*TCA*
avg. size	**47.4**	82.7	104.0	86.6	86.7	98.1
avg. time	284.6	52.7	42.1	377.9	357.2	52.7

*To save space, we use '*Short*' to denote '*CAmpactor-Short*'.

Table 2 in the paper; all the execution times includes that of *SamplingCA*
CAmpactor-Short uses the running time of *TCA* as cutoff time

Evaluation (cont'd)

- RQ1: comparison with SOTA competitors
- Result: *CAmpactor* is also efficient

	CAmpactor	*Short**	*SamplingCA*	*AutoCCAG*	*FastCA*	*TCA*
avg. size	**47.4**	82.7	104.0	86.6	86.7	98.1
avg. time	284.6	52.7	42.1	377.9	357.2	52.7

*To save space, we use '*Short*' to denote '*CAmpactor-Short*'.

Table 2 in the paper; all the execution times includes that of *SamplingCA*
CAmpactor-Short uses the running time of *TCA* as cutoff time

Evaluation (cont'd)

- RQ1: comparison with SOTA competitors
- Result: *CAmpactor* is also efficient
  - *CAmpactor* outperforms other competitors even if it is only allowed to run ~10 more seconds in average

	CAmpactor	*Short**	*SamplingCA*	*AutoCCAG*	*FastCA*	*TCA*
avg. size	**47.4**	82.7	104.0	86.6	86.7	98.1
avg. time	284.6	52.7	42.1	377.9	357.2	52.7

*To save space, we use '*Short*' to denote '*CAmpactor-Short*'.

Table 2 in the paper; all the execution times includes that of *SamplingCA*
CAmpactor-Short uses the running time of *TCA* as cutoff time

Evaluation (cont'd)

- RQ1: comparison with SOTA competitors
- Result: *CAmpactor* is also efficient
  - *CAmpactor* outperforms other competitors even if it is only allowed to run ~10 more seconds in average

	CAmpactor	*Short**	*SamplingCA*	*AutoCCAG*	*FastCA*	*TCA*
avg. size	**47.4**	82.7	104.0	86.6	86.7	98.1
avg. time	284.6	52.7	42.1	377.9	357.2	52.7

*To save space, we use '*Short*' to denote '*CAmpactor-Short*'.

Table 2 in the paper; all the execution times includes that of *SamplingCA*
CAmpactor-Short uses the running time of *TCA* as cutoff time

Summary

- *CAmpactor*: an novel and effective algorithm dedicated for PCA optimization
- Target: overcoming the scalability problem of PCA optimization
  - Tacking the hindering issue with forced patching
  - Mitigating the cycling issue by strengthening forbidden strategy
- Result: *CAmpactor* is effective and also moderately efficient

- For our paper and our tool: check the entry of our paper on FSE’23 website!

Summary

- *CAmpactor*: an novel and effective algorithm dedicated for PCA optimization
- Target: overcoming the scalability problem of PCA optimization
  - Tacking the hindering issue with forced patching
  - Mitigating the cycling issue by strengthening forbidden strategy
- Result: *CAmpactor* is effective and also moderately efficient

- For our paper and our tool: check the entry of our paper on FSE'23 website!

Thank you for listening! 😀

Back Up

RQ2: Ablation Study

- Alt-1: *CAmpactor* minus test case level forbidden strategy
- Alt-2: *CAmpactor* minus test case level forbidden strategy, plus single value level forbidden strategy
- Alt-3: *CAmpactor* minus forced patching

Table 3: Average size and average running time of *CAmpactor* and all its alternative versions over all instances.

	CAmpactor	*Alt-1*	*Alt-2*	*Alt-3*
avg. size	**47.4**	79.8	54.3	98.6
avg. time	284.6	67.3	278.7	46.1

RQ4: Generality of *CAmpactor*

- Alt-A/F/T: using *CAmpactor* to optimize the output PCA from *AutoCCAG*/*FastCA*/*TCA*

Table 7: Average size and average running time of *AutoCCAG*, *FastCA*, *TCA*, *Alt-A*, *Alt-F* and *Alt-T* over all instances.

	AutoCCAG	*Alt-A*	*FastCA*	*Alt-F*	*TCA*	*Alt-T*
avg. size	86.6	47.4	86.7	**47.3**	98.1	**47.3**
avg. time	377.9	611.1	357.2	591.5	52.7	293.4