Optimizing Selection of Competing Features via Feedback-Directed Evolutionary Algorithms

Presenter: Tian Huat Tan

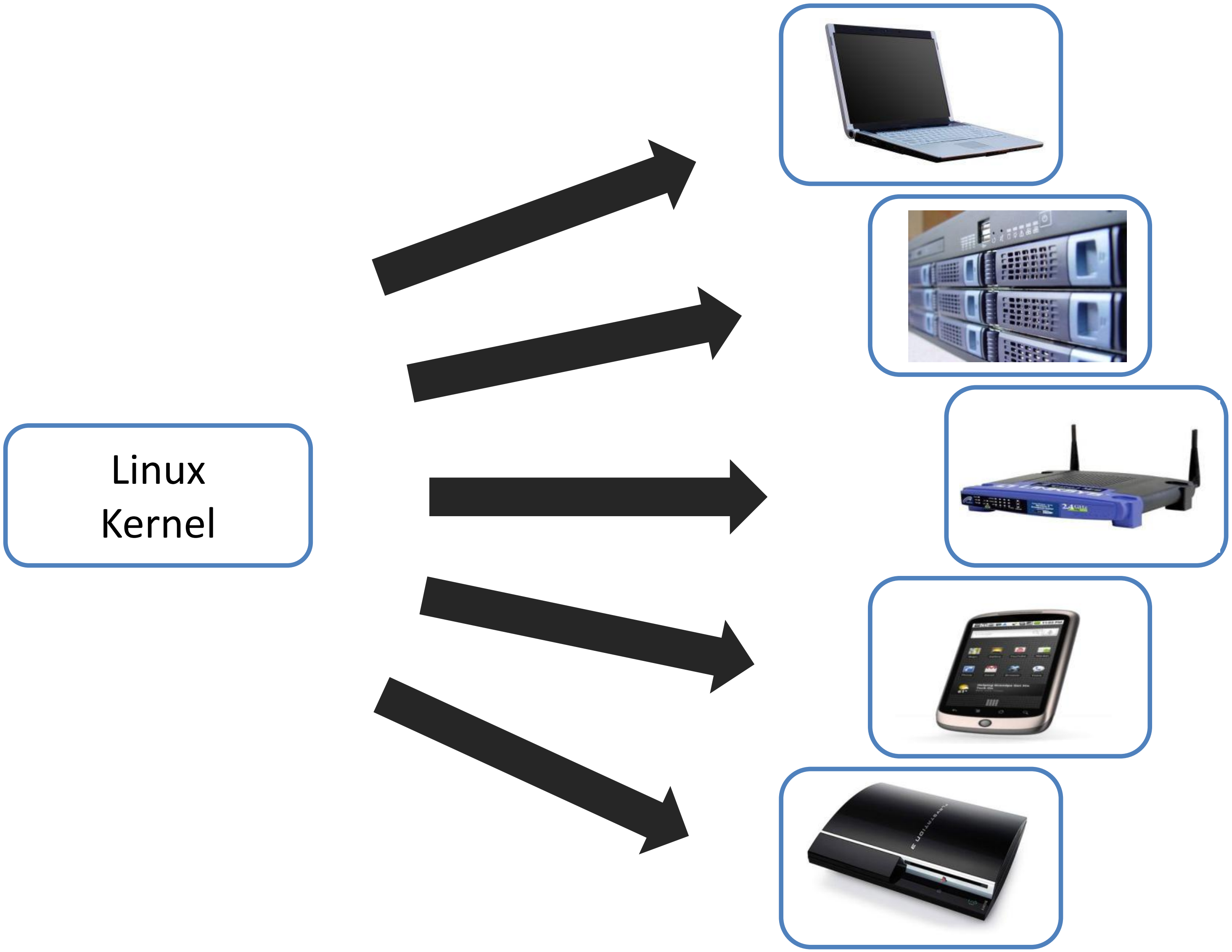
Tian Huat Tan[1] , Yinxing Xue[2] , Manman Chen[3] ,
Jun Sun[1] , Yang Liu[2] , Jin Song Dong[3]

[1]Singapore University of Technology and Design,
[2]Nanyang Technological University,
[3]Singapore National University of Singapore

Software Product Line

- A Software Product Line (SPL) is a family of products designed to take advantage of their common features and specified variations
- The ultimate goals is to mitigate production costs and improve the quality from the perspective of a customer.

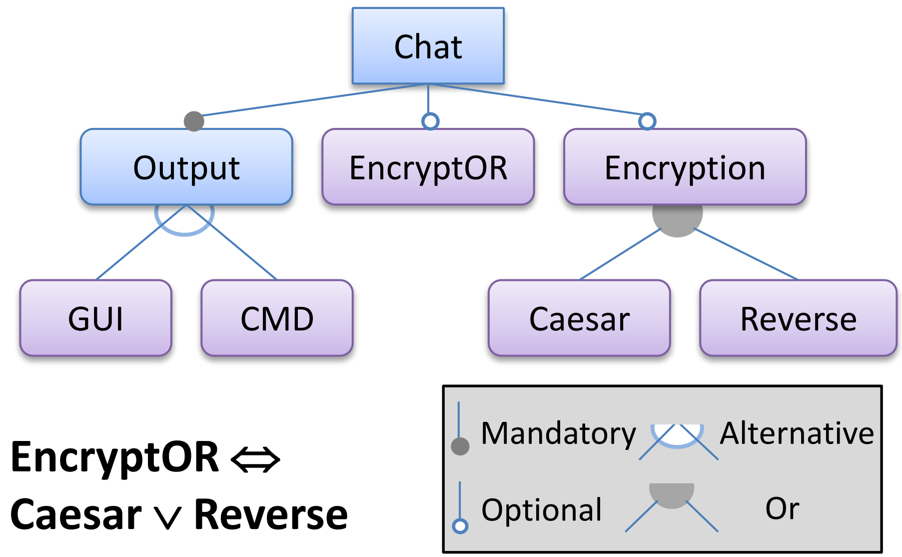
Example: Linux Kernel



Feature Model

Visual representation of software product line in tree format to facilitate reasoning and understanding.

Feature Model – Java Chat Model



- Each feature could associate with quality attributes such as response time and cost
- A valid feature set/product = {Chat, Output, GUI}

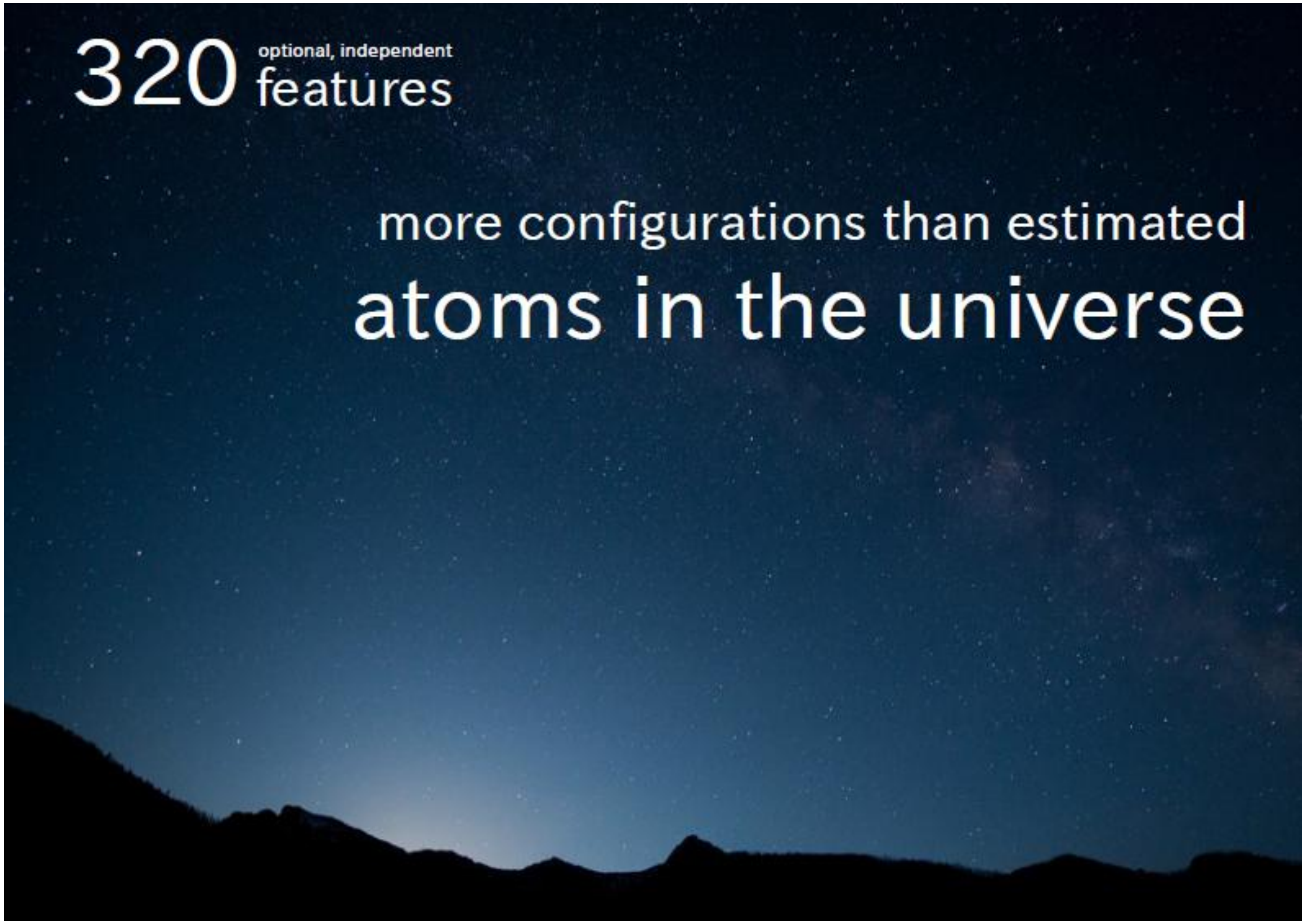
Two Main Objectives

Given a feature model, we want to generate products that:

- **Conform to feature model**
  - Select a set of features that complies with the feature model
- **Satisfy user preferences**
  - Optimizes the quality attributes (e.g., response time, cost) of products according to user preferences.

This is known as **Optimal Feature Selection Problem**

Challenge 1: Exponentially Many Configurations



Challenge 2: Conflicting User Preferences

- Example:
  - Maximize number of features
  - Minimize Costs

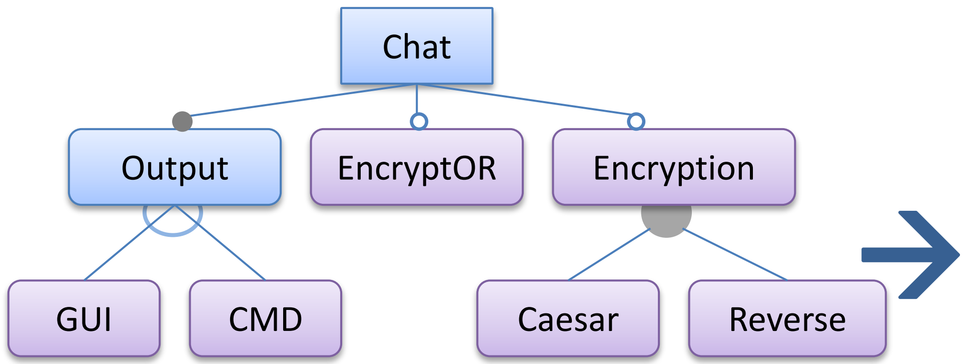
Approach

- Feature model → Logical Constraints
- Multi-objective Evolutionary Algorithm (Pruning + Feedback Directed)

Approach

- Feature model → Logical Constraints
- Multi-objective Evolutionary Algorithm (Pruning + Feedback Directed)

Feature Model → Logical Constraints



EncryptOR ⇔ Caesar ∨ Reverse

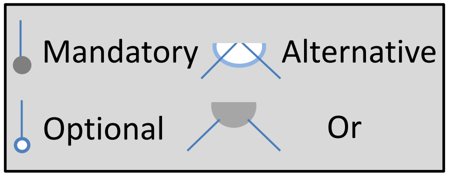
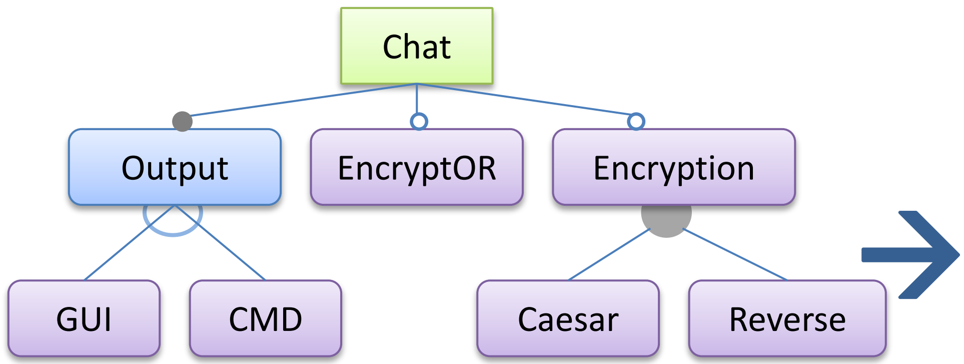


Table: Constraints of JCS	
c(1)	Chat
c(2)	Output ⇔ Chat
c(3)	EncryptOR ⇒ Chat
c(4)	Encryption ⇒ Chat
c(5)	(GUI ∨ CMD) ⇔ Output
c(6)	¬ (GUI ∧ CMD)
c(7)	(Caesar ∨ Reverse) ⇔ Encryption
c(8) Cross Tree Constraint	EncryptOR ⇔ (Caesar ∨ Reverse)

Feature Model → Logical Constraints



EncryptOR ⇔ Caesar ∨ Reverse

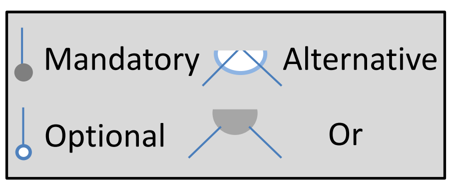
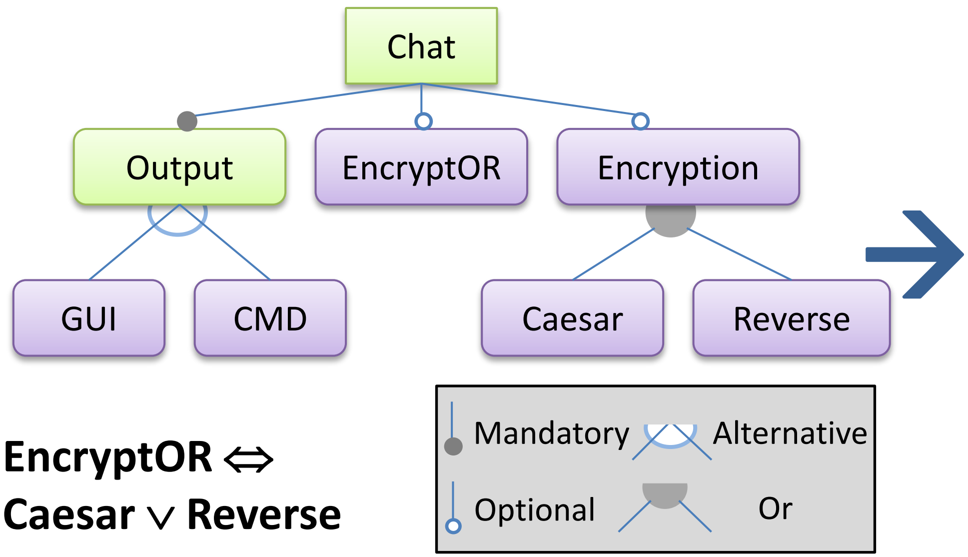


Table: Constraints of JCS

c(1)	Chat
c(2)	Output ⇔ Chat
c(3)	EncryptOR ⇒ Chat
c(4)	Encryption ⇒ Chat
c(5)	(GUI ∨ CMD) ⇔ Output
c(6)	¬ (GUI ∧ CMD)
c(7)	(Caesar ∨ Reverse) ⇔ Encryption
c(8) Cross Tree Constraint	EncryptOR ⇔ (Caesar ∨ Reverse)

Feature Model → Logical Constraints



EncryptOR ⇔ Caesar ∨ Reverse

Table: Constraints of JCS	
c(1)	Chat
c(2)	Output ⇔ Chat
c(3)	EncryptOR ⇒ Chat
c(4)	Encryption ⇒ Chat
c(5)	(GUI ∨ CMD) ⇔ Output
c(6)	¬ (GUI ∧ CMD)
c(7)	(Caesar ∨ Reverse) ⇔ Encryption
c(8) Cross Tree Constraint	EncryptOR ⇔ (Caesar ∨ Reverse)

Approach

- Feature model → Logical Constraints
- Multi-objective Evolutionary Algorithm (Pruning + Feedback Directed)

Approach

- Feature model → Logical Constraints
- Multi-objective Evolutionary Algorithm (Pruning + Feedback Directed)

Multiple Objectives

- **Correctness**:
  minimize the number of violated constraints of the feature model.
- **Richness of features:**
  minimize the number of features that are not selected.
- **Cost**:
  minimize the total cost.
- **Feature used before**:
  minimize the number of features that have not been used before.
- **Defects**:
  minimize the number of known defects.

Multi-objective Optimization Problem

- A *k*-objective optimization problem:

  Minimize $Obj(F) = (Obj_1(F), Obj_2(F), \ldots, Obj_k(F))$ (1)

- $Obj(F_1)$ is smaller than $Obj(F_2)$ or F_1 dominates F_2 in Equation (1), if

  $\forall i: Obj_i(F_1) \leq Obj_i(F_2) \land \exists j: Obj_j(F_1) < Obj_j(F_2)$

  where $i, j \in \{1,\ldots,k\}$

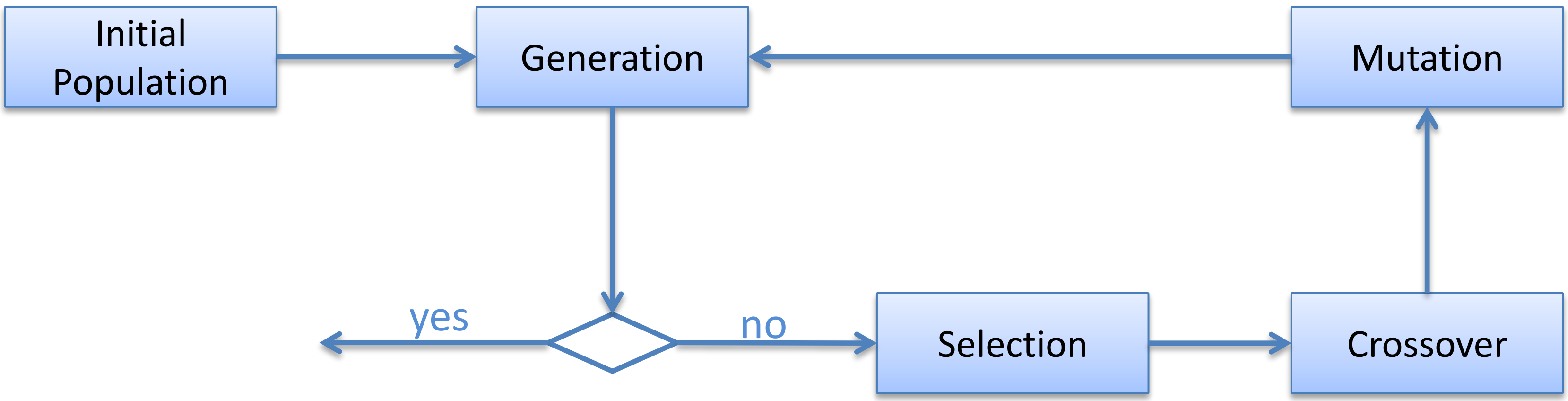
- F is called a Pareto-optimal solution if it is not dominated by any other F'.

Approach

- Feature model → Logical Constraints
- Multi-objective Evolutionary Algorithm (Pruning + Feedback Directed)

Evolutionary Algorithm (EA)

Finding optimal solutions based on mechanisms inspired by biologic evolution



Examples of Multi-objective EA

- Examples: IBEA, NSGA-II, ssNSGA-II, MOCell
- Based on the dominating criteria they used

Approach

- Feature model → Logical Constraints
- Multi-objective Evolutionary Algorithm
  (Pruning + Feedback Directed)

Key Insight

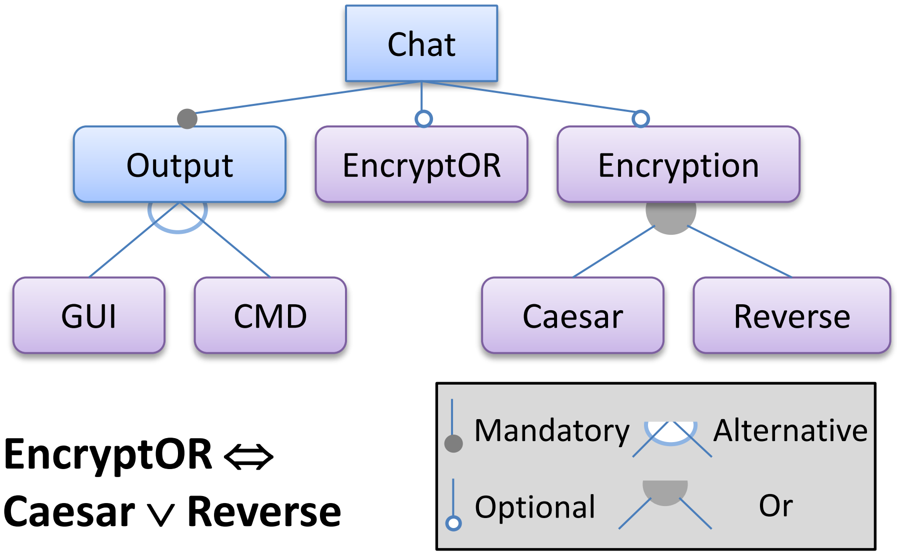
- Some features are always used or never used at all (Prune them)
- The crossover and mutation operation is generic – they are not adaptive to optimal feature selection (Feedback-directed EA)

Approach

- Feature model → Logical Constraints
- Multi-objective Evolutionary Algorithm (Pruning + Feedback Directed)

Prune Common and Dead Features

- Common Features: Feature set shared by all derived products ({Chat, Output})
  - $\neg SAT(\, fea \wedge \neg f \,)$
- Dead Features : Feature that must not be used by all derived products
  - $\neg SAT(fea \wedge f \,)$



EncryptOR ⇔ Caesar ∨ Reverse

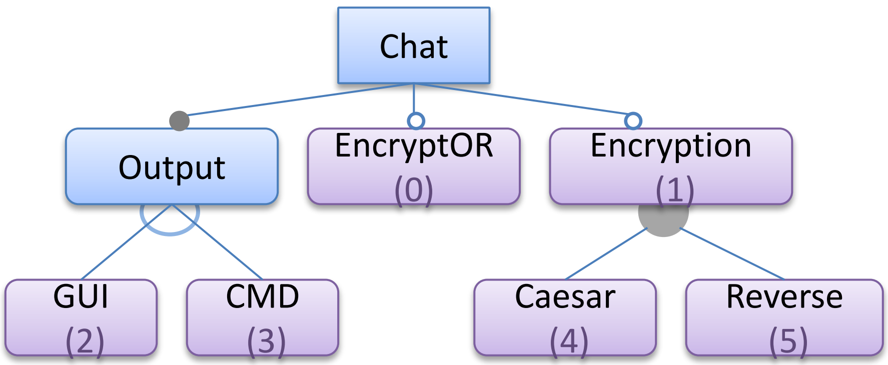
Approach

- Feature model → Logical Constraints
- Multi-objective Evolutionary Algorithm (Pruning + Feedback Directed)

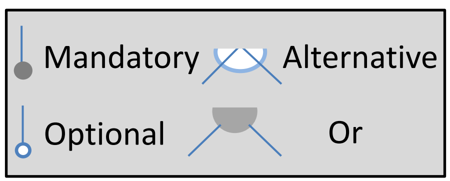
Feedback-directed EA

- What is the feedback?
  They are selected features that do not comply to the feature model.
- We use feedback to improve solutions of next generation
- Feedback is incorporated by means of crossover and mutation operation

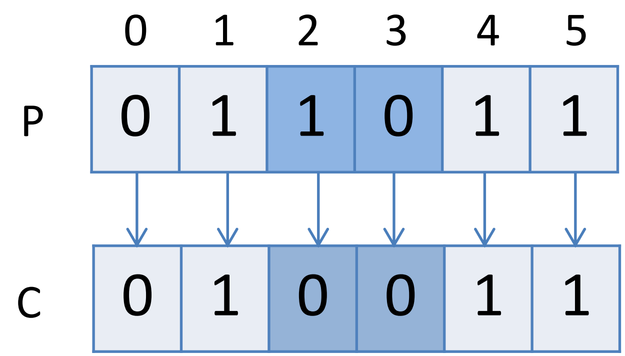
Mutation



EncryptOR ⇔
Caesar ∨ Reverse

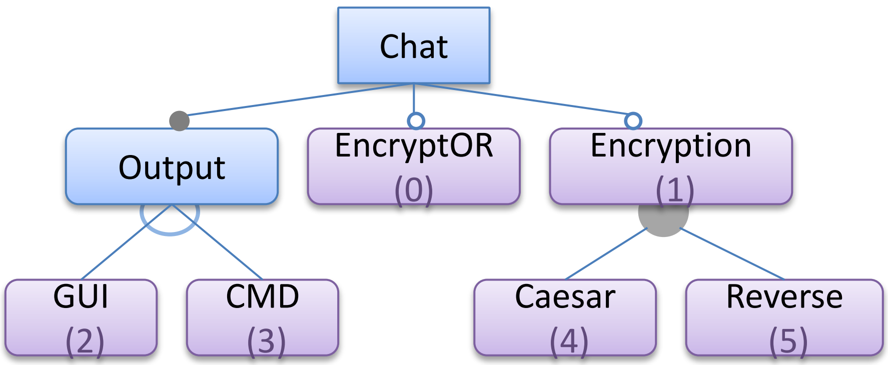


{Encryption, GUI, Caesar, Reverse}

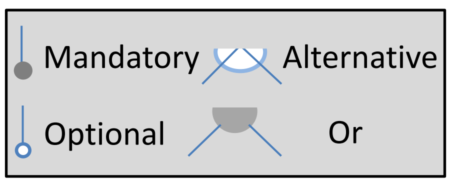


Mutation Rate 0.01

Feedback-Directed Mutation



EncryptOR ⇔
Caesar ∨ Reverse



{Encryption, GUI, Caesar, Reverse}

0	1	2	3	4	5
0	1	1	0	1	1

EncryptOR (0) ⇔ **Caesar** (4) ∨ **Reverse** (5)

Mutation Rate 0.0000001
Error Mutation Rate 1

Evaluation: Benchmark

- SPLOT (Software Product Line Online Tools)
  - An online repository of product line
- LVAT (Linux Variability Analysis Tools)
  - Reversed Engineered from big projects like Linux kernel and eCos operating system.

Evaluation: Metrics

- **Percentage of Correctness** (%Correct):
  The solutions that are valid.

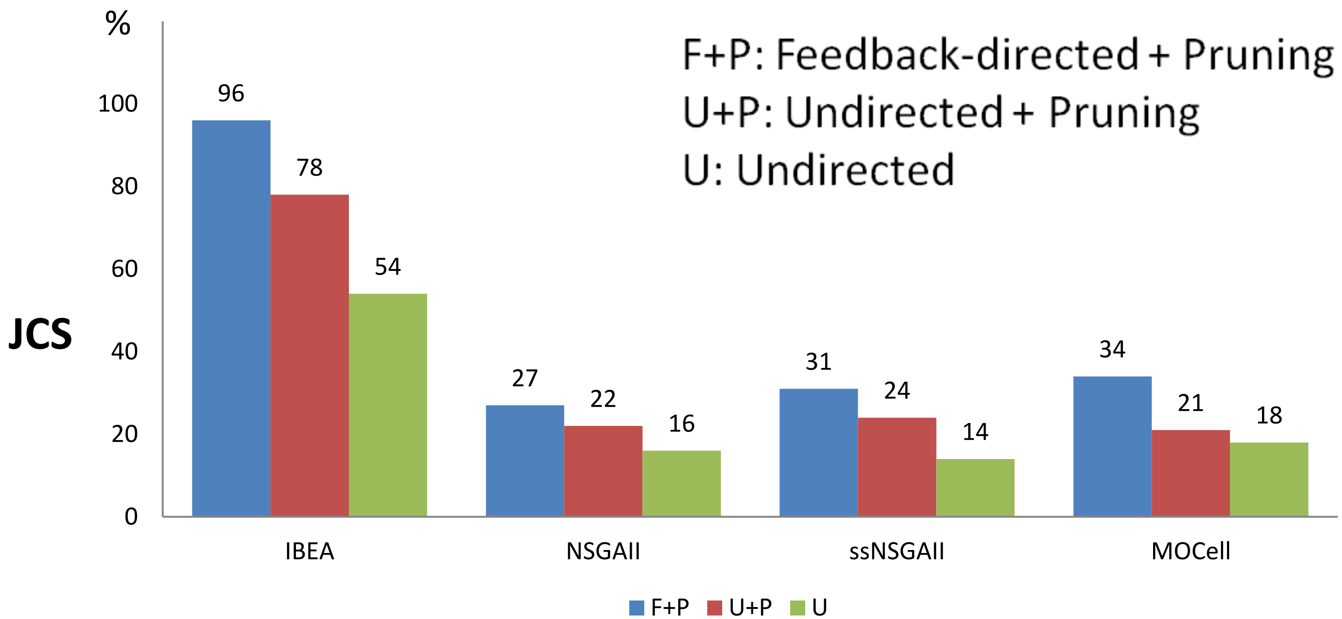
- **Hypervolume**:
  Hypervolume of the solution set is the volume of the region that is dominated by solution.

Evaluation: Objectives

- **Correctness**:
  minimize the number of violated constraints of the feature model.
- **Richness of features:**
  minimize the number of features that are not selected.
- **Cost**:
  minimize the total cost.
- **Feature used before**:
  minimize the number of features that have not been used before.
- **Defects**:
  minimize the number of known defects.

Evaluation: SPLOT

25000 evaluations



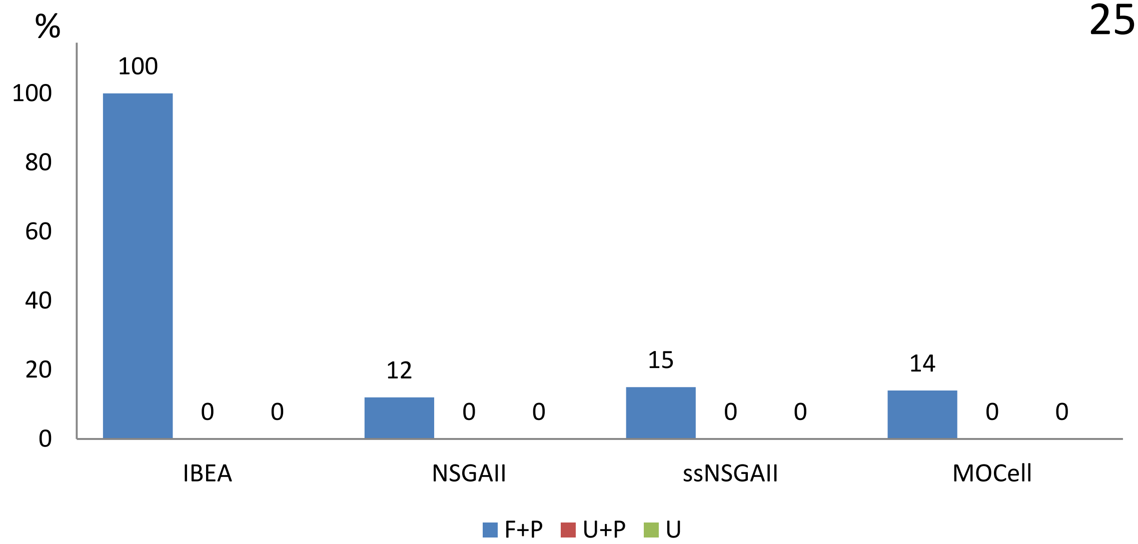
- IBEA outperformed other methods
- F+P is better than U+P; U+P is better than U
- State-of-the-art [1,2]: Somewhere between U+P and U

[1]A. S. Sayyad, J. Ingram, T. Menzies, and H. Ammar. Scalable product line conguration: A straw to break the camel's back. In ASE, 2013.
[2]A. S. Sayyad, T. Menzies, and H. Ammar. On the value of user preferences in search-based software engineering: a case study in software product lines. In ICSE, pages 492-501, 2013.

Evaluation: SPLOT

25000 evaluations

E-shop



- For U+P for IBEA, it achieved 46% of correctness for 3.25 hours.
- For F+P for IBEA, it achieved 100% of correctness by just 6.9 seconds.

Evaluation: Linux Kernel (Seeding Method)

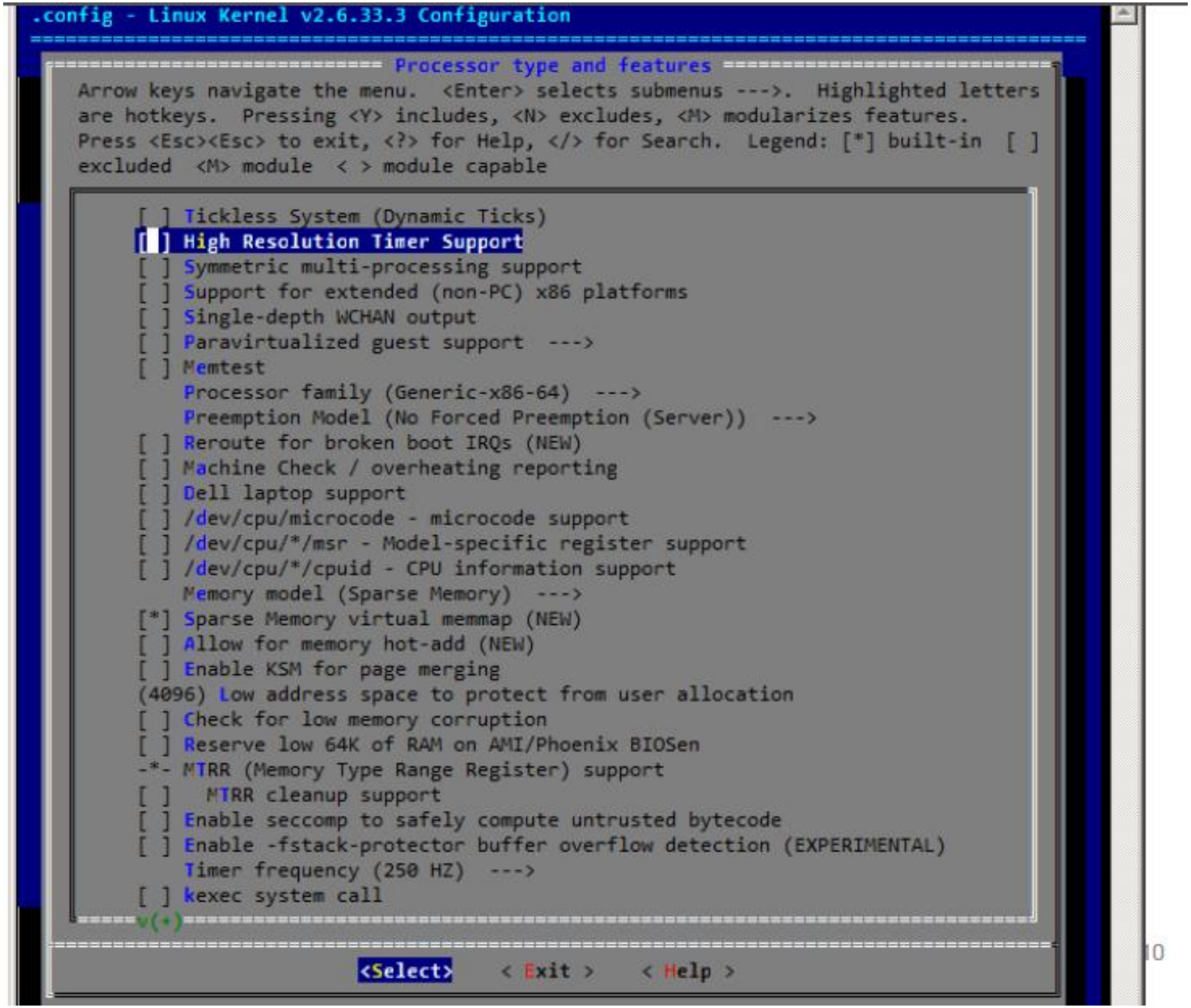
- Linux Kernel has 6888 features
- IBEA with two objectives is used to generate the seed.
- U+P spends a total 4 hours of execution time for 36 correct solutions.
- F+P uses less than 40 seconds to get 36 correct solutions

Conclusion

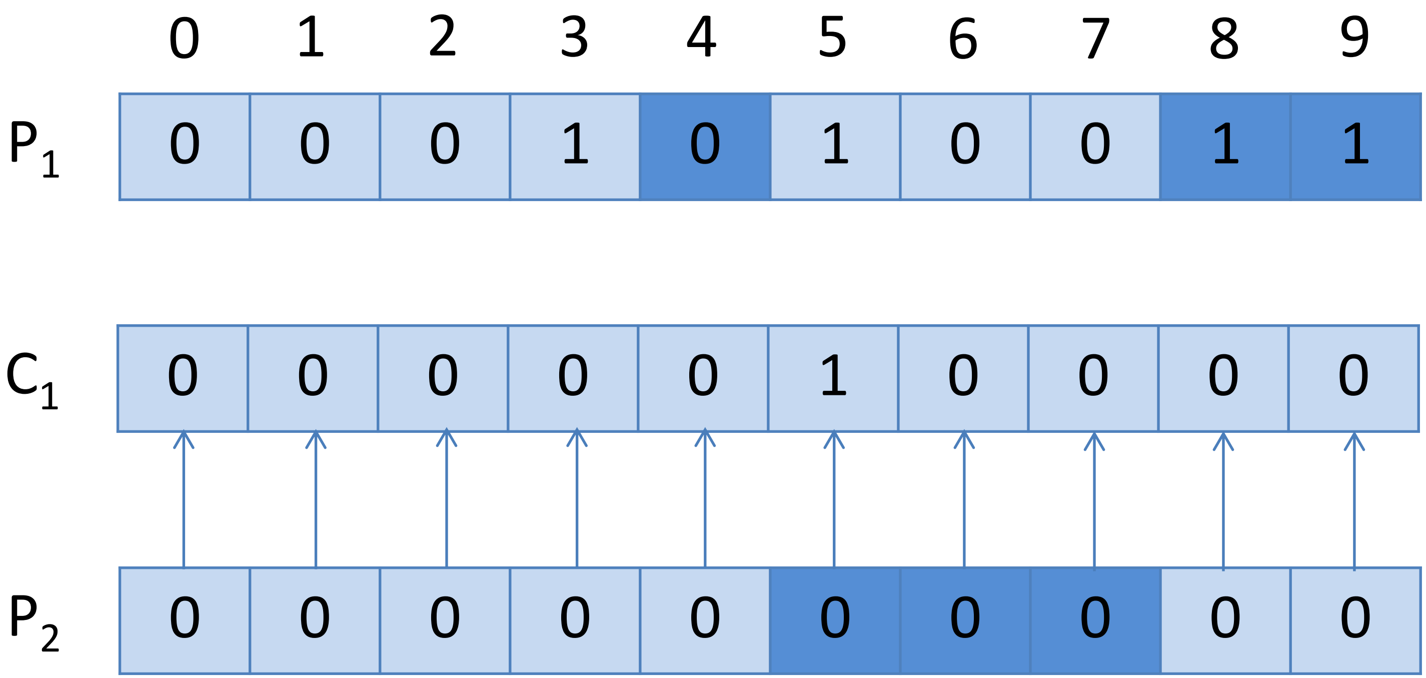
- **Generality** - Our technique improves common EAs in optimal feature selection.
- **Faster Convergence** – Our technique allows efficient and effective findings of optimal features.

Thank you!

Email: ttianhuat@gmail.com


.config - Linux Kernel v2.6.33.3 Configuration
Processor type and features
Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [ ] excluded <M> module < > module capable
[ ] Tickless System (Dynamic Ticks)
[ ] High Resolution Timer Support
[ ] Symmetric multi-processing support
[ ] Support for extended (non-PC) x86 platforms
[ ] Single-depth WCHAN output
[ ] Paravirtualized guest support --->
[ ] Memtest
Processor family (Generic-x86-64) --->
Preemption Model (No Forced Preemption (Server)) --->
[ ] Reroute for broken boot IRQs (NEW)
[ ] Machine Check / overheating reporting
[ ] Dell laptop support
[ ] /dev/cpu/microcode - microcode support
[ ] /dev/cpu/*/msr - Model-specific register support
[ ] /dev/cpu/*/cpuid - CPU information support
Memory model (Sparse Memory) --->
[*] Sparse Memory virtual memmap (NEW)
[ ] Allow for memory hot-add (NEW)
[ ] Enable KSM for page merging
(4096) Low address space to protect from user allocation
[ ] Check for low memory corruption
[ ] Reserve low 64K of RAM on AMI/Phoenix BIOSen
-*- MTRR (Memory Type Range Register) support
[ ] MTRR cleanup support
[ ] Enable seccomp to safely compute untrusted bytecode
[ ] Enable -fstack-protector buffer overflow detection (EXPERIMENTAL)
Timer frequency (250 HZ) --->
[ ] kexec system call
v(+)
<Select> < Exit > < Help >

Feedback-Directed Crossover



P_1: ✕ c(13): Encryption_OR (4) ⇔ Caesar (8) ∨ Reverse (9)

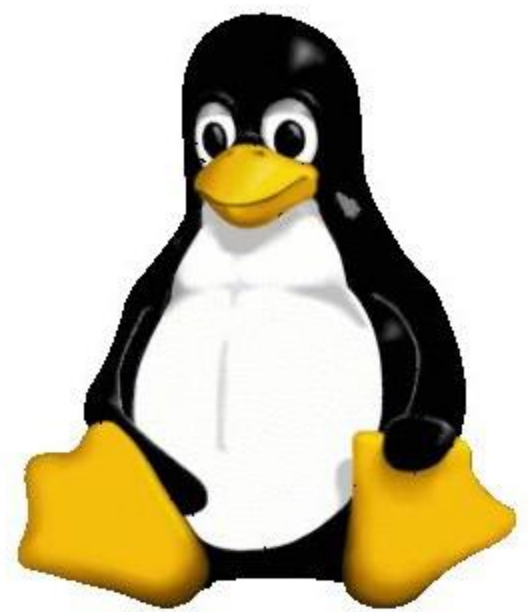
P_2 : ✕ c(8): (GUI (5) ∨ CMD (6) ∨ GUI2 (7)) ⇔ Output

Example: Linux Kernel

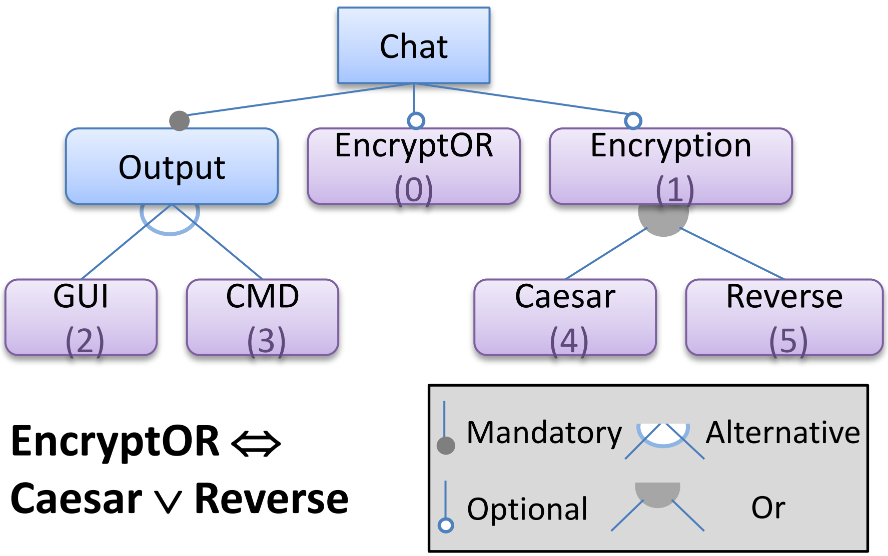
- ~6,000,000 Lines of C code
- Highly configurable

❑ > 10,000 configuration options! (x86, 64bit, …)

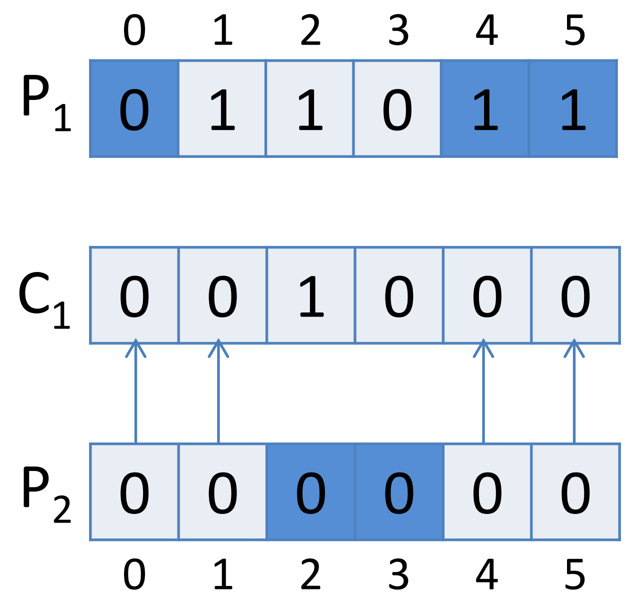
❑ Most source code is “optional”



Feedback-Directed Crossover



EncryptOR ⇔ Caesar ∨ Reverse



P_1: ✘ **EncryptOR** (0) ⇔ **Caesar** (4) ∨ **Reverse** (5)

P_2 : ✘ **GUI** (2) ∨ **CMD** (3) ⇔ **Output**

Evaluation: Feature Attribute

- Cost $\in$ R:
  the cost incurred to use the feature.
- Used_Before $\in$ {0,1}:
  whether the feature has been used before.
- Defects $\in$ Z:
  the number of defects known in the feature.