Using Machine Learning to Predict the Sequences of Optimization Passes

Presenters: Anurag Bangera, Chirag Bangera, Jonhan Chen, Richard Wang

Paper Authors: Laith H. Alhasnawy, Esraa H. Alwan, and Ahmed B. M. Fanfakh

December 1st, 2023

Table of Contents

Context

- What is a pass?
  - Analysis step
  - Transformation step
- Optimization passes
  - Dead code elimination, Loop Unrolling, etc.
- In LLVM: Optimizer - “**opt**”
  - Specify which optimization passes to use
  - Arrange the order of optimization passes

Context

- Phase / Pass order:
  - The sequence of passes run on a program
- “The **Phase (pass) Order** Problem”
  - Orderings of passes can impact effectiveness of each pass
  - Interdependencies between passes
  - No universal optimal ordering
- Pass order is manually tuned — static
  - Not always ideal! Could be sub-optimal, causing slow down.

Significance

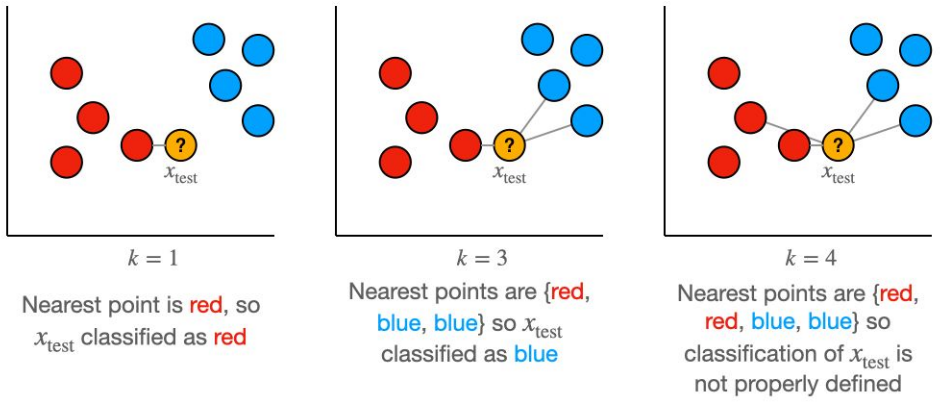
- **Optimization sequence space:** set of all valid pass orders
    - For reference, there are over 10^64 pass orders given 50 passes.
        - LLVM: ~10 to ~70 passes
        - GCC: ~30 to ~100 passes
    - Impossibly large to search through

Purpose of the Paper

- **The Big Idea**:
  - Dynamically auto-tuning optimization sequences without testing all pass orders
- Main Topics:
  - **Machine Learning**: Use K-NN algorithm to build a prediction scheme.
  - **Feature Pass**: Extract static features for each program.
  - **Reduction Algorithm**: Improve the resulting sequences from KNN model.

Step 1: K-Nearest Neighbors Model

- *"Non-parametric, supervised learning classifier"*
  - **Non-parametric**: No assumptions on parameters
  - **Supervised Learning**: input parameters and corresponding "correct" value train a model
  - **Classifier**: Identifies which of a set of categories an observation belongs to
- Uses proximity to make classifications or predictions
  - Distance function



Step 2: Feature Extraction

- **Feature**: a “parameter” for machine learning models to classify based on
- Many possible relevant features related to optimizations
- Researchers used: **instruction count**
  - Add instructions
  - Alloca instructions
  - And instructions
  - Etc…
- Resulted in 39 total features, each numerical

Step 3: Model Training

Goal: Generate a semi-optimized pass for each program

- It’s really hard to find the actual optimized path for each algorithm
- Use a greedy search to be able to get a good result for training

Algorithm 1: Finding Optimization Sequence

Input: benchmark programs, optimization passes.
Output: optimization sequence of passes for each program (*collectionSequencArray*).

```
1: For i=1 to end of benchmark programs
2:      Features extraction (program before executing optimization passes)
3: End For i
4: For i=1 to end of benchmark programs
5:For j=1 to end of optimization passes
6:              Apply (–scalarrepl pass)
7:              Features extraction (program after executing each optimization pass)
8:      End For j
9: End For i
10: For i=1 to end of benchmark programs
11:New_program←benchmark program
12:     For j  = 1 to length  New_program
13:               Use KNN to classify New_program[j] for two closest programs by its
                    features.
14:               Choose best two optimization passesfrom two closest programs.
15:Save two chosenoptimizationpasses in SequencArray.
16:          Save two optimization passes as new programs in Temp_new_programs.
17:End For j
18          New_program = Temp_new_programs
19:While (!End optimization passes) goto step 12
20:  Save SequencArray in collectionSequencArray
21: End For i
22:      Return collectionSequencArray
```

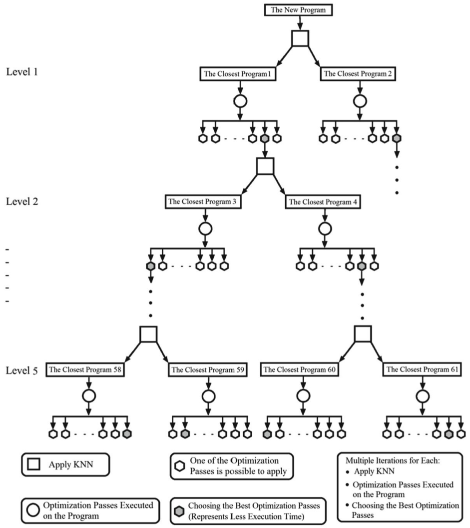
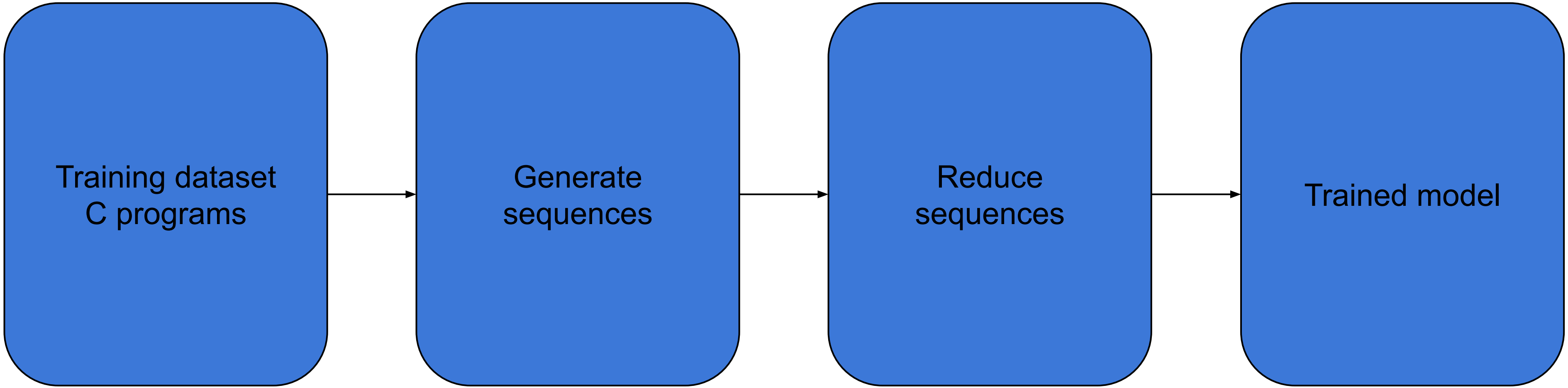
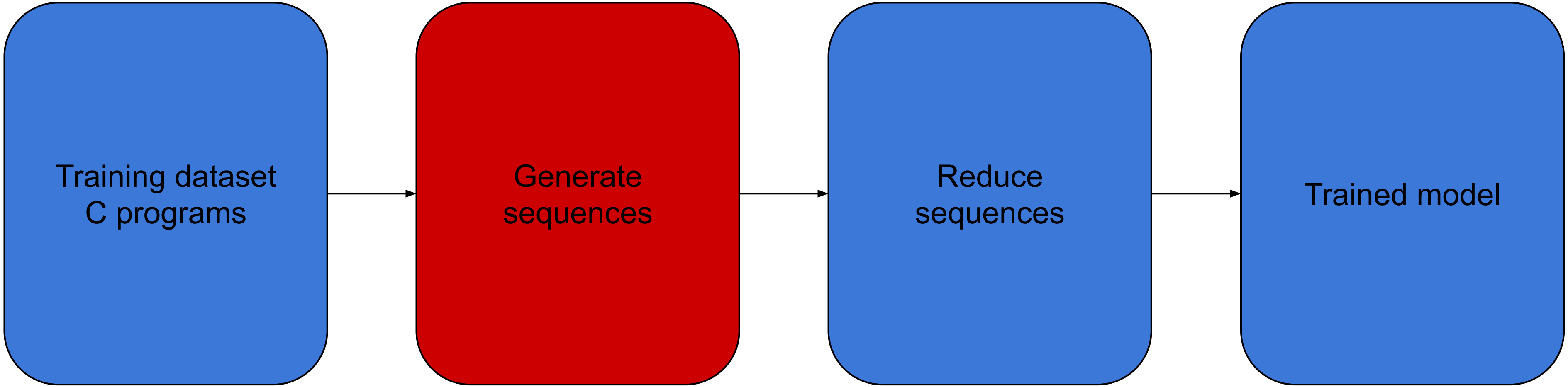


Fig. 2. The scheme of finding optimization sequence

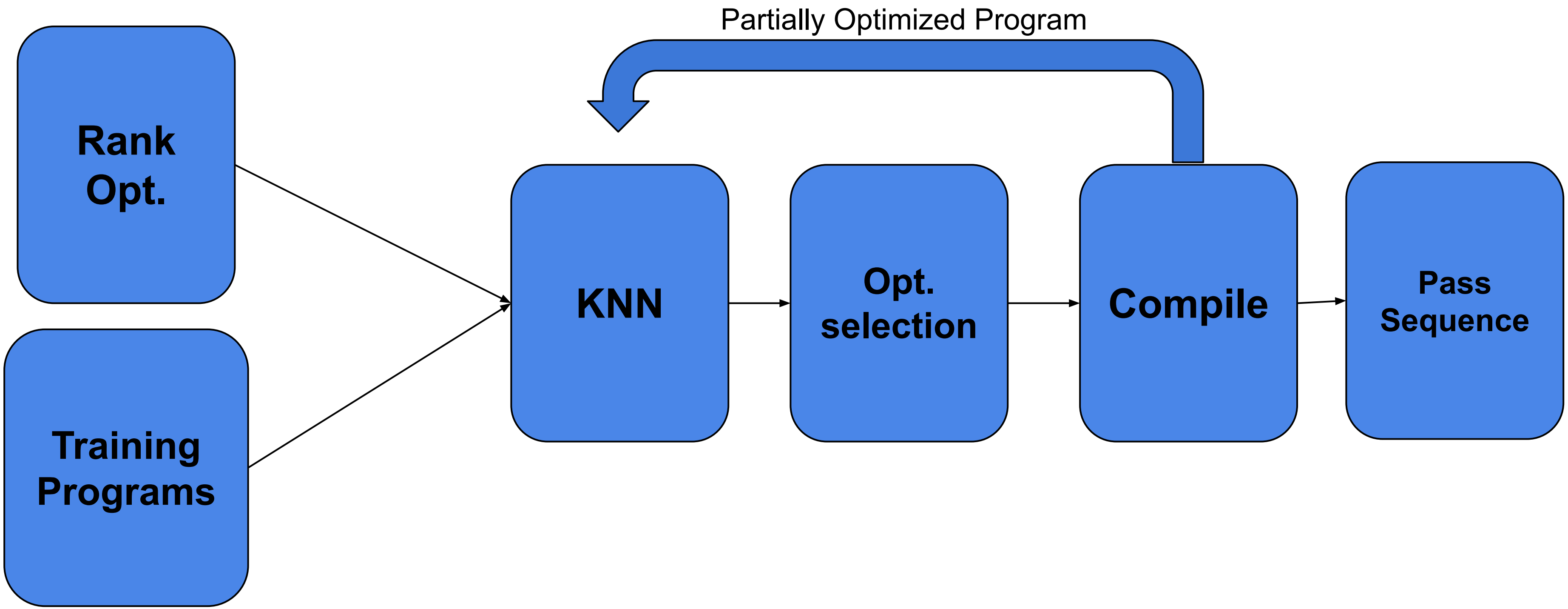
Model Training Overview



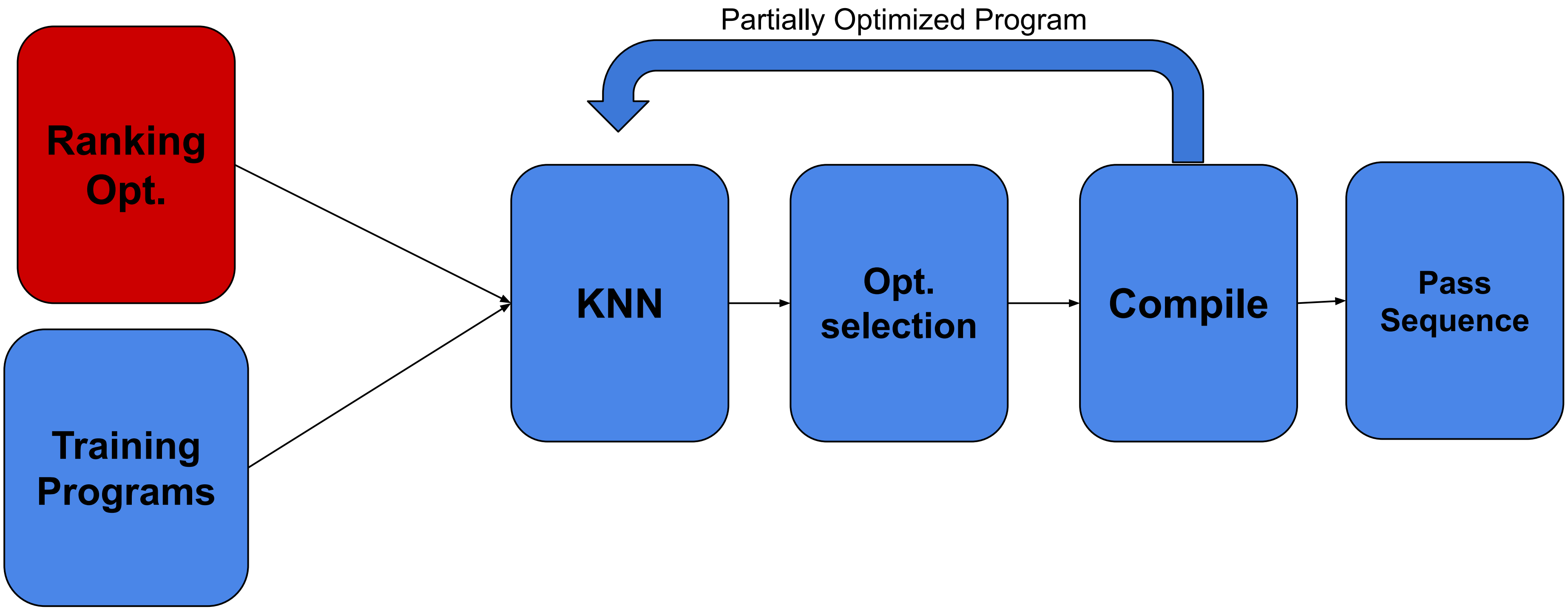
Model Training Overview



Generating Pass Sequences



Generating Pass Sequences



Ranking Optimizations

A.c
1. Const Prop
2. DCE
3. Loop Inv
4. Loop unroll
...

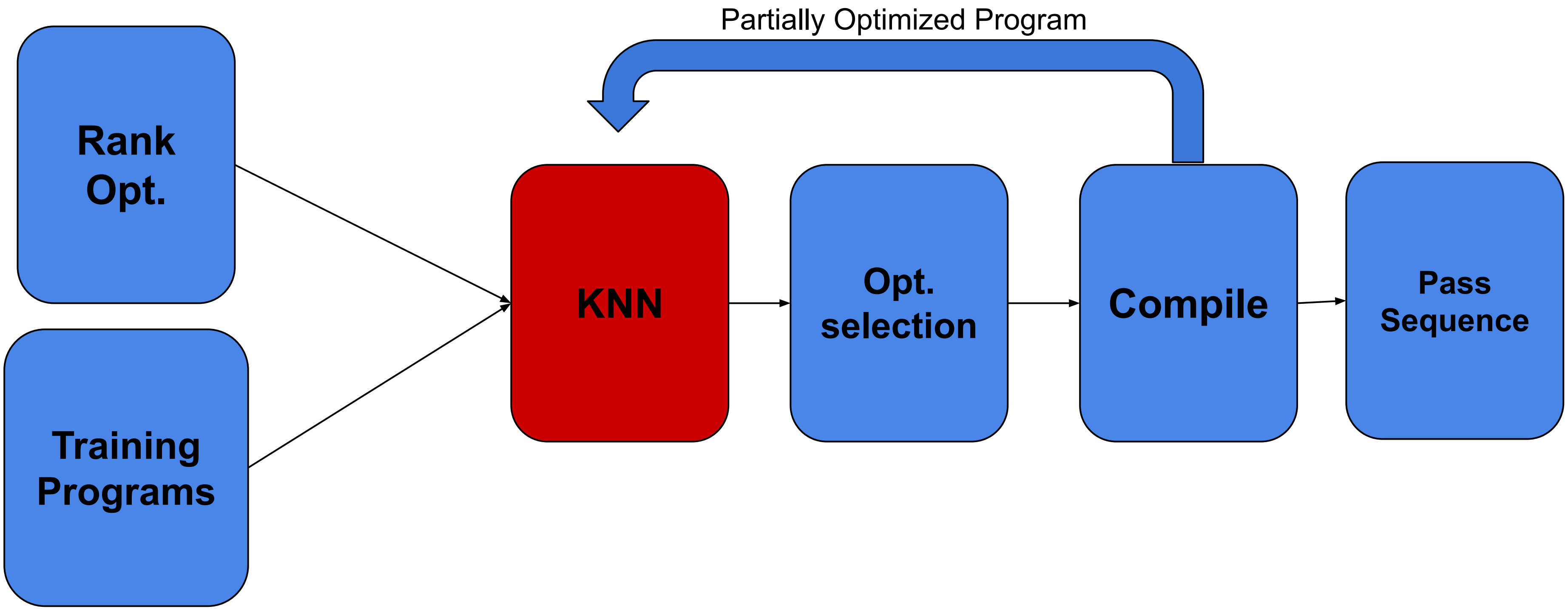
B.c
1. Const Prop
2. DCE
3. Loop Inv
4. Loop unroll
...

C.c
1. Const Prop
2. DCE
3. Loop Inv
4. Loop unroll
...
...

D.c
1. Const Prop
2. DCE
3. Loop Inv
4. Loop unroll
...

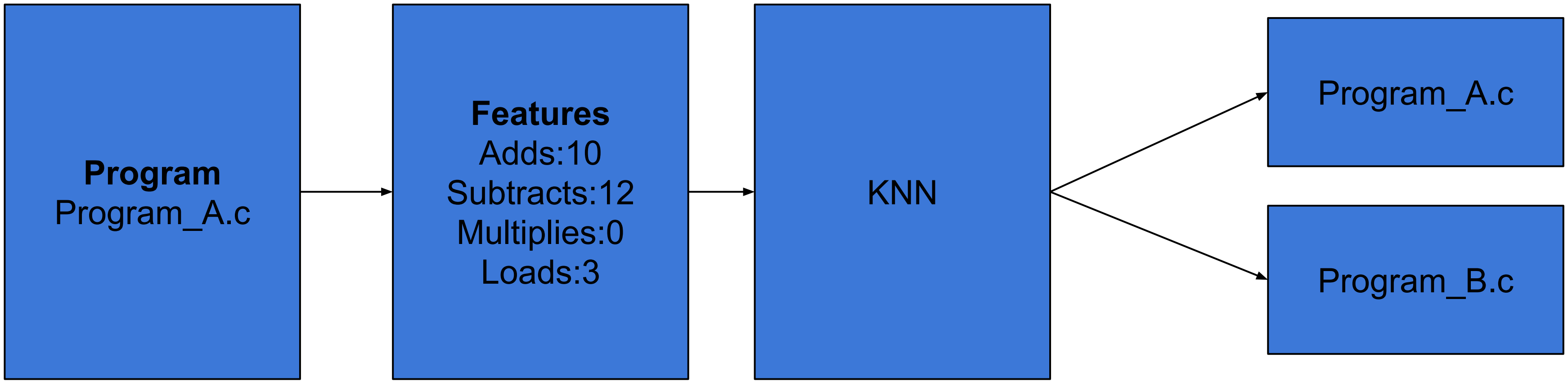
- Run each optimization on each program individually
- This does not capture order, but rather the best optimizations on the initial program, so we need to do more

Generating Pass Sequences

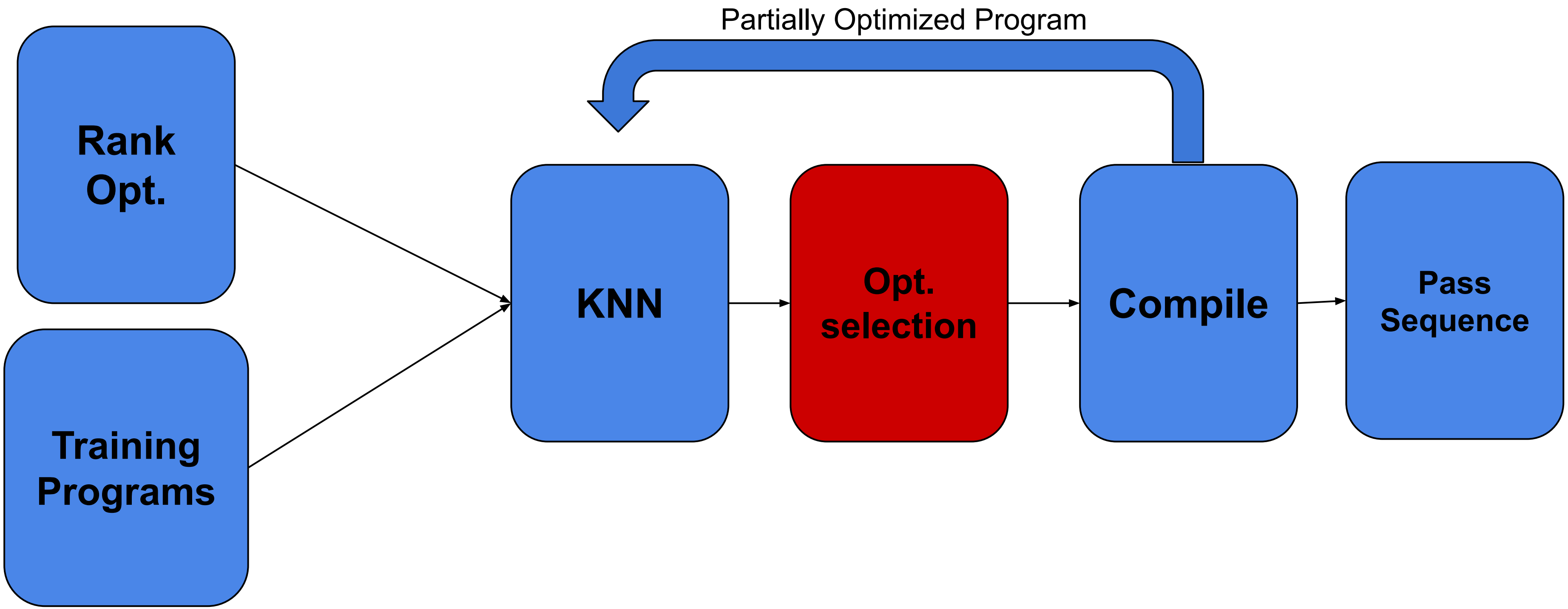


KNN

We run feature extraction and our KNN model to identify the closest two programs

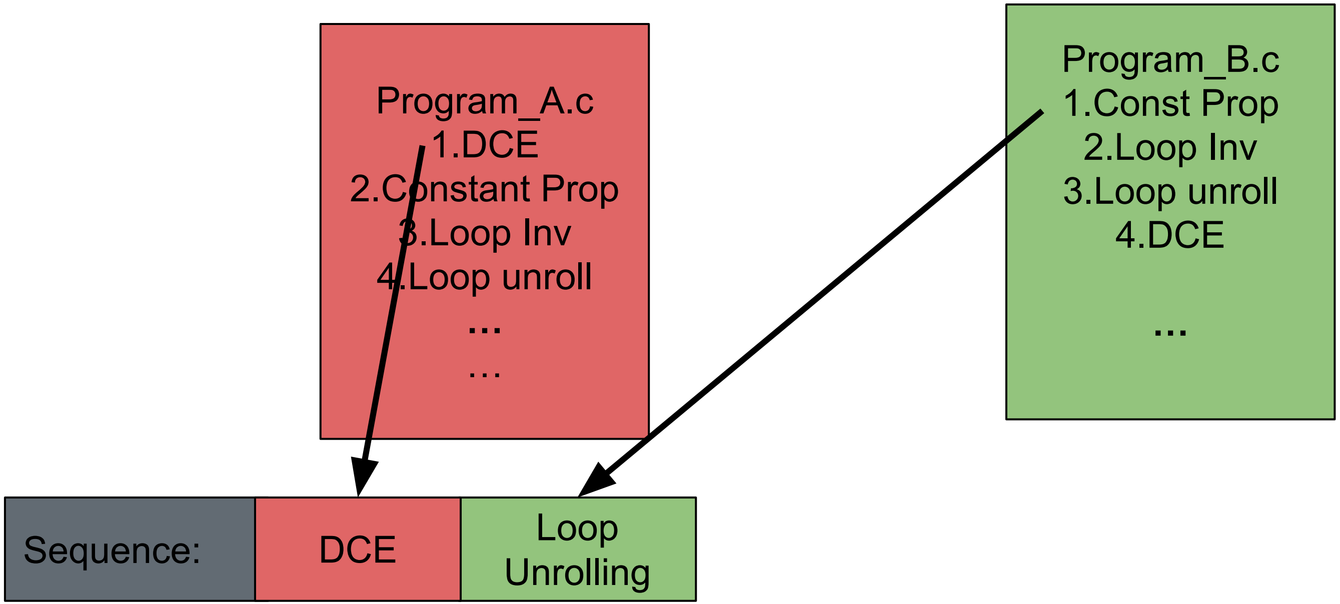


Generating Pass Sequences

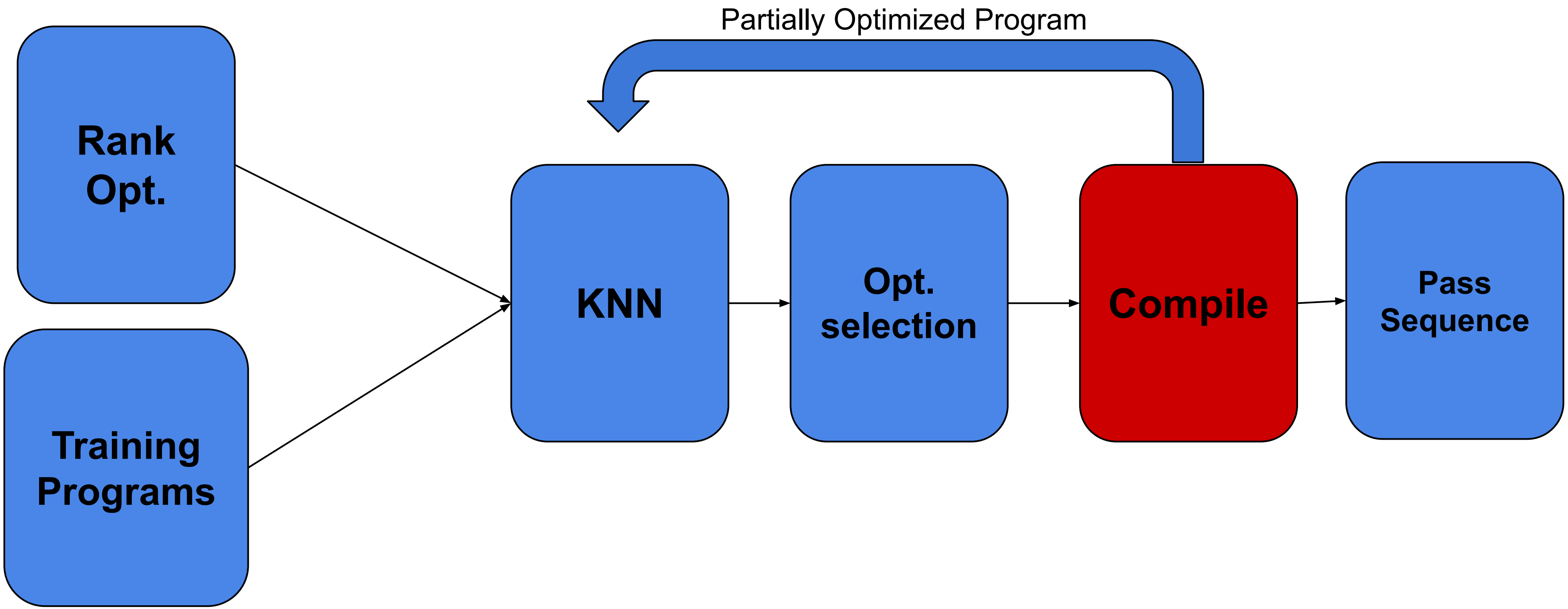


Optimization Selection

-We then add the highest unused optimizations from the programs to our sequence

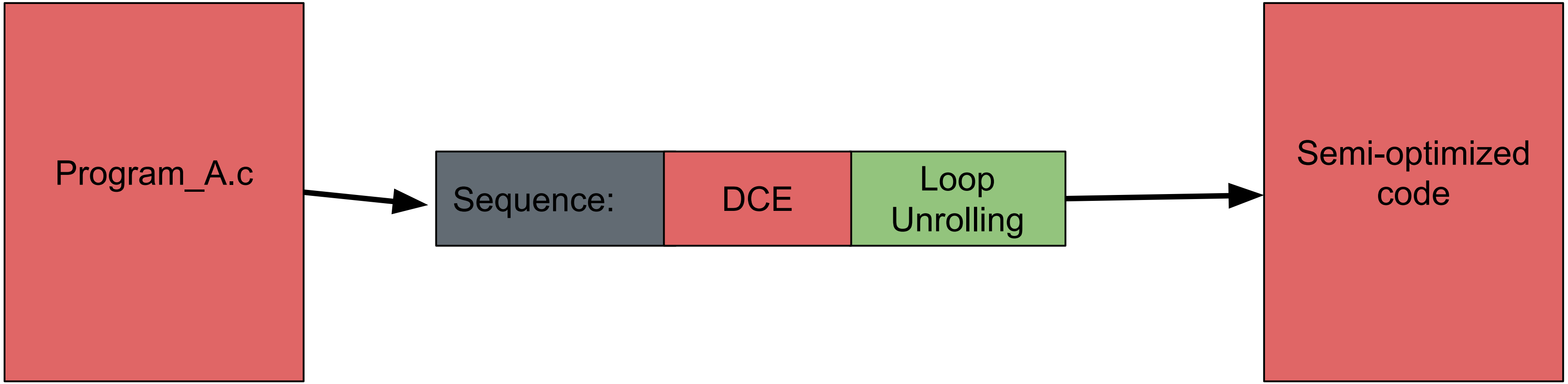


Generating Pass Sequences

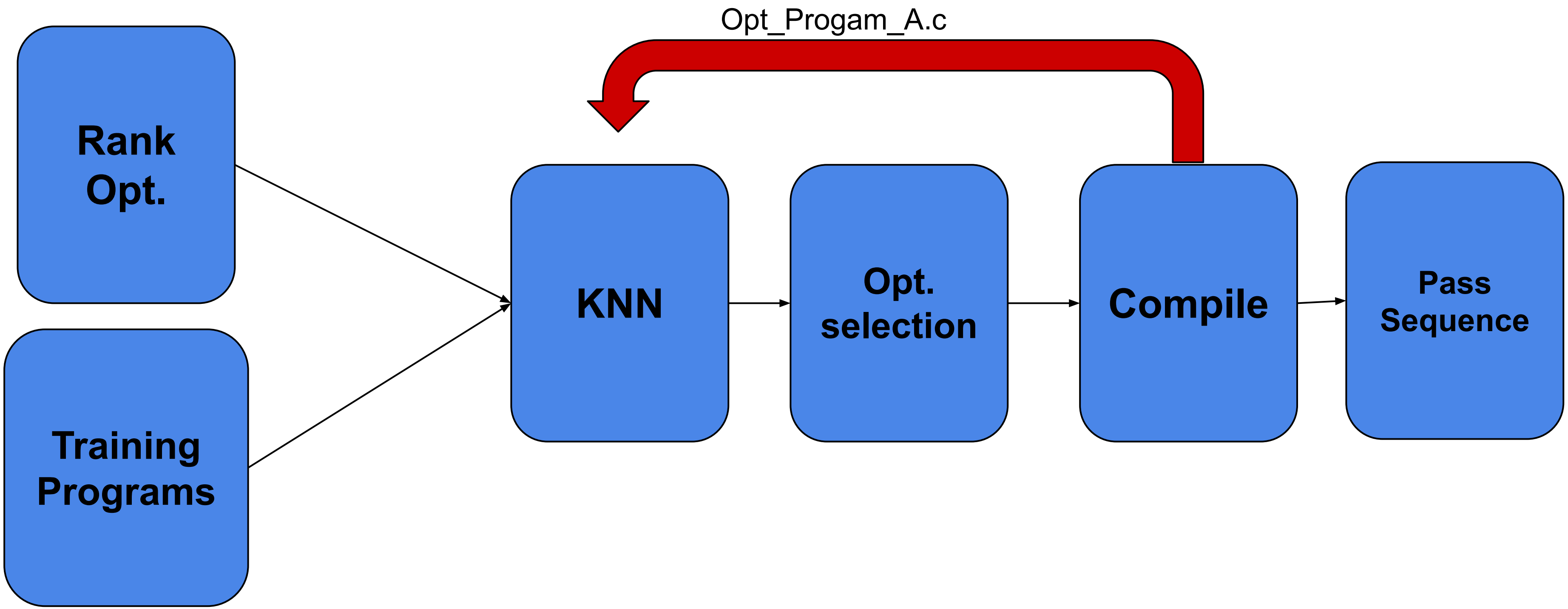


Compile

- We compile the current program with the optimization pass sequence

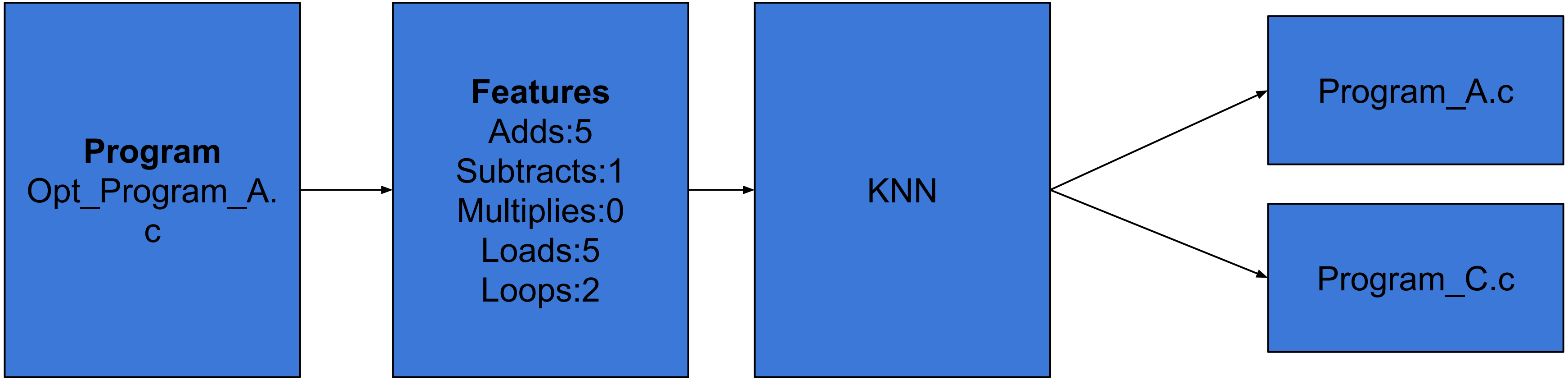


Repeat!


Opt_Progam_A.c
Rank Opt.
Training Programs
KNN
Opt. selection
Compile
Pass Sequence

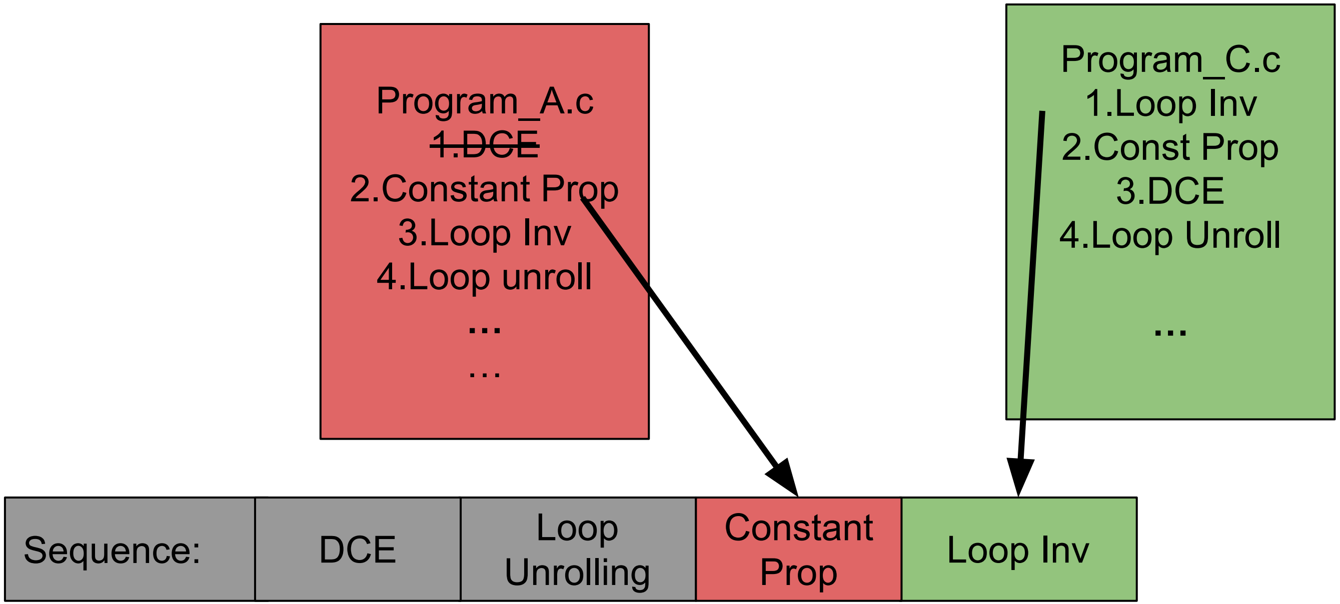
KNN

We run feature extraction and our KNN model to identify the closest two programs



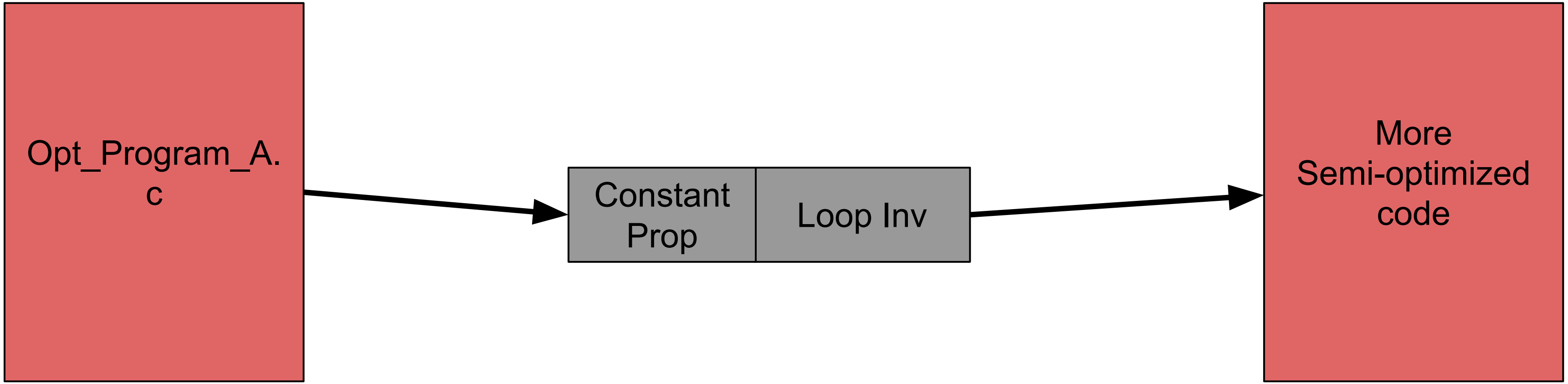
Optimization Selection

-We then add the highest unused optimizations from the programs to our sequence

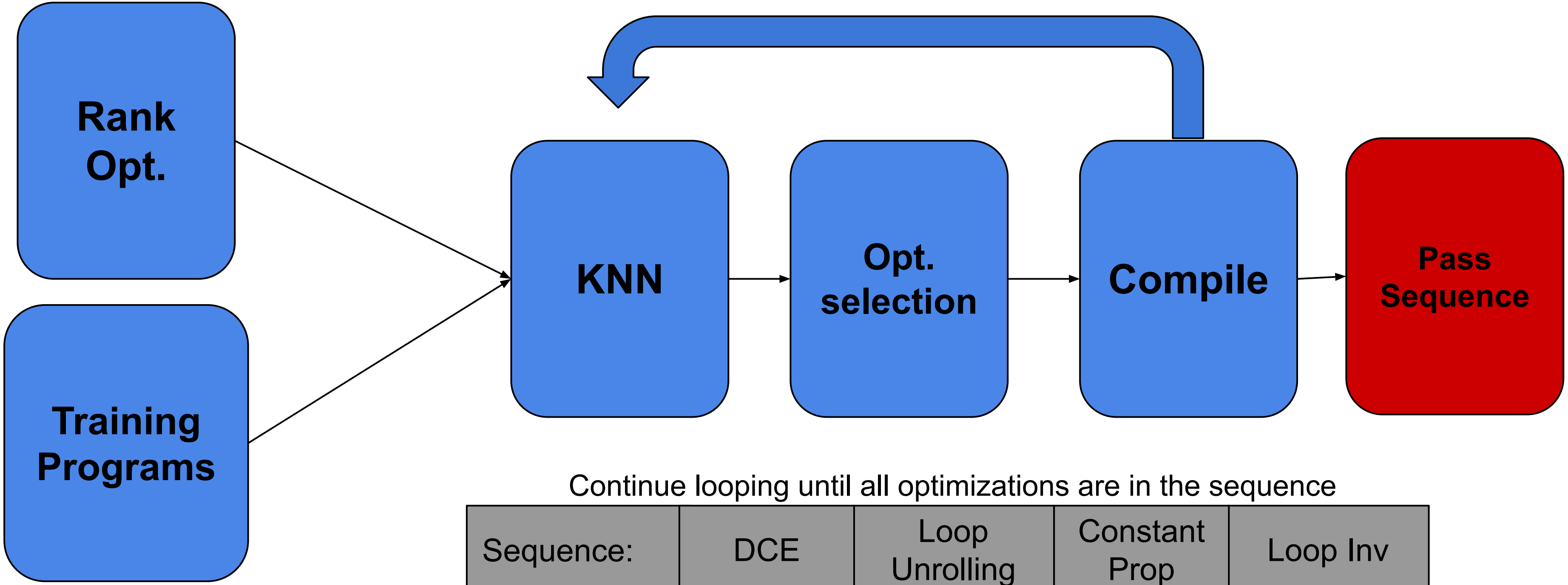


Compile

- We compile the current program with the optimization pass sequence



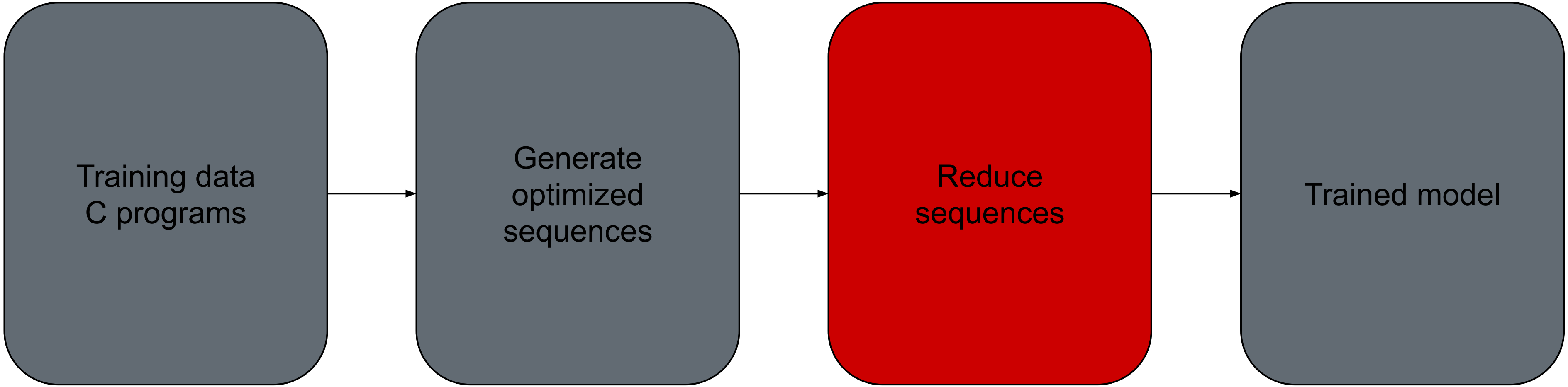
Pass Sequence



Continue looping until all optimizations are in the sequence

Sequence:	DCE	Loop Unrolling	Constant Prop	Loop Inv

Training: Reduction



Step 3: Reduction Algorithm

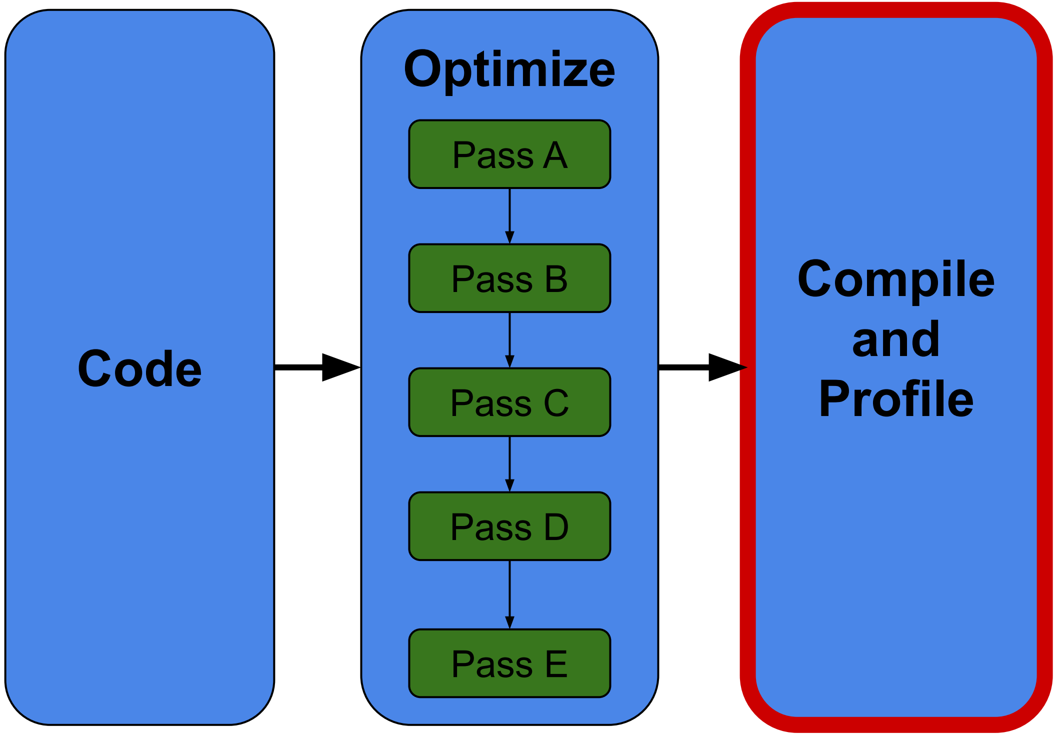
- **Motivation:** Some optimization passes can actually increase the runtime of the program

- **Solution:** Run a reduction pass to remove any non-improvement passes from the training data

Algorithm 2:Reduction Optimization Sequence

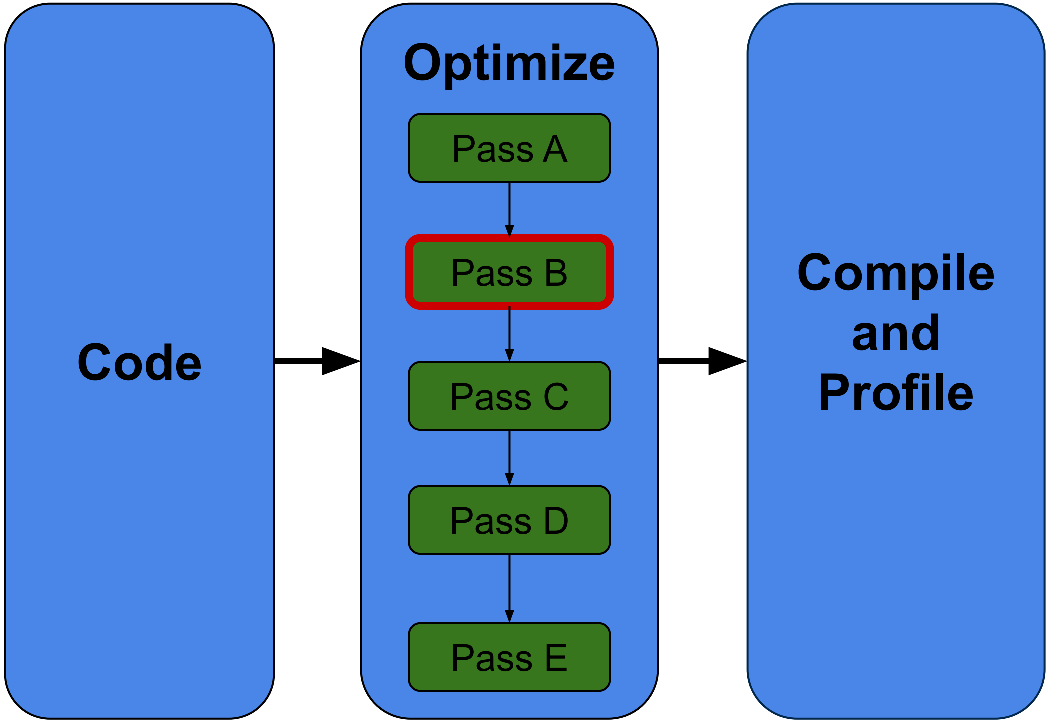
```
Input:prog_Opt_seq by using KNN.
Output: Best_Opt_seq.
Execution time ← execution (prog_Opt_seq)
I← 1
While I< length of  prog_Opt_seq
        Temp_prog_Opt_seq← Delete (prog_Opt_seq, I)
        Best_Execution time ← execution(Temp_prog_Opt_seq)
    If Best_Execution time < Execution time
        Execution time ← Best_Execution time
Best_Opt_seq←Temp_prog_Opt_seq
I←0
End if
        I←I+1
End  while
        Return  Best Opt_seq
```

Training: Reduction



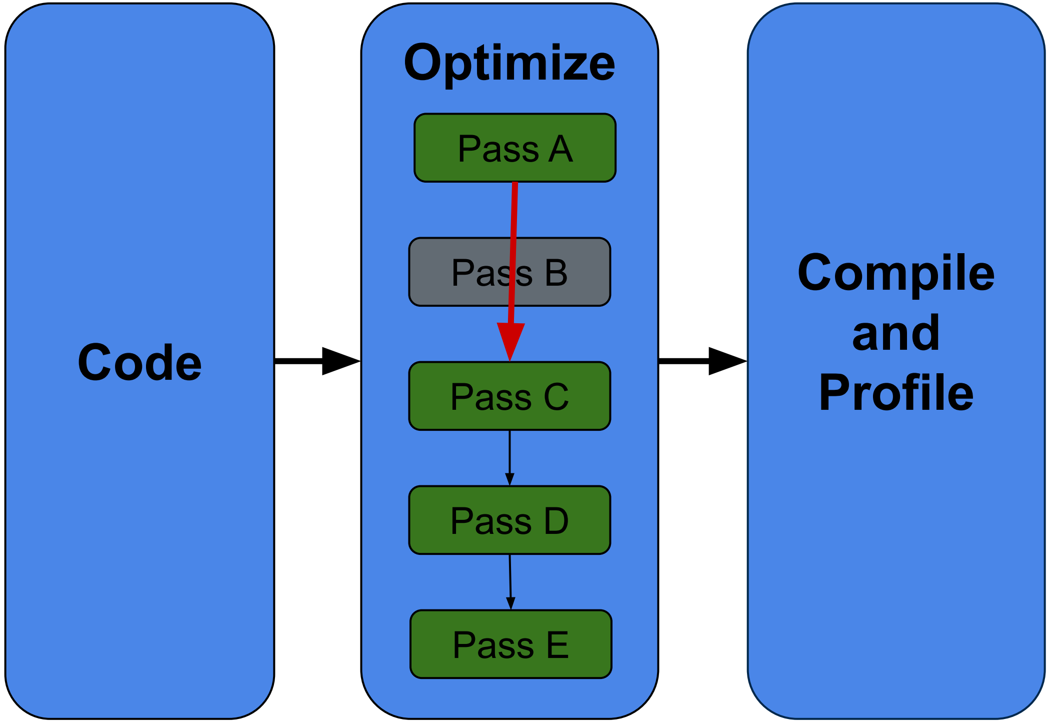
1. **Establish Baseline time**
2. Select a pass to test
3. Deactivate Pass
4. Recompile code
5. Profile new executable
6. Compare

Training: Reduction



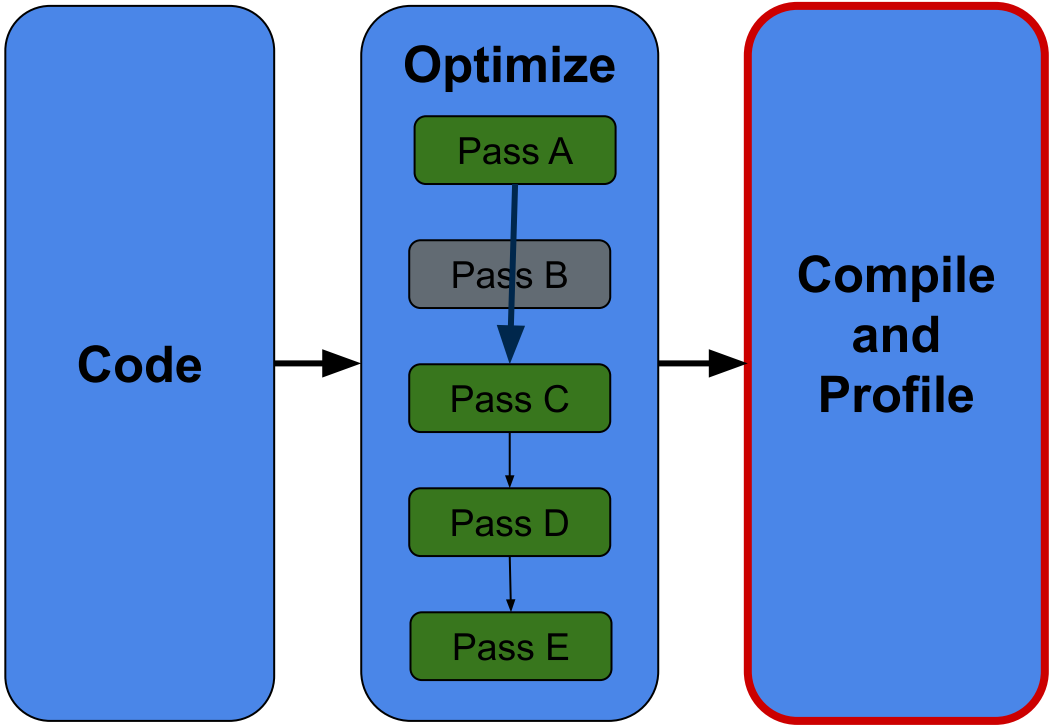
1. Establish Baseline time
2. **Select a pass to test**
3. Deactivate Pass
4. Recompile code
5. Profile new executable
6. Compare

Training: Reduction



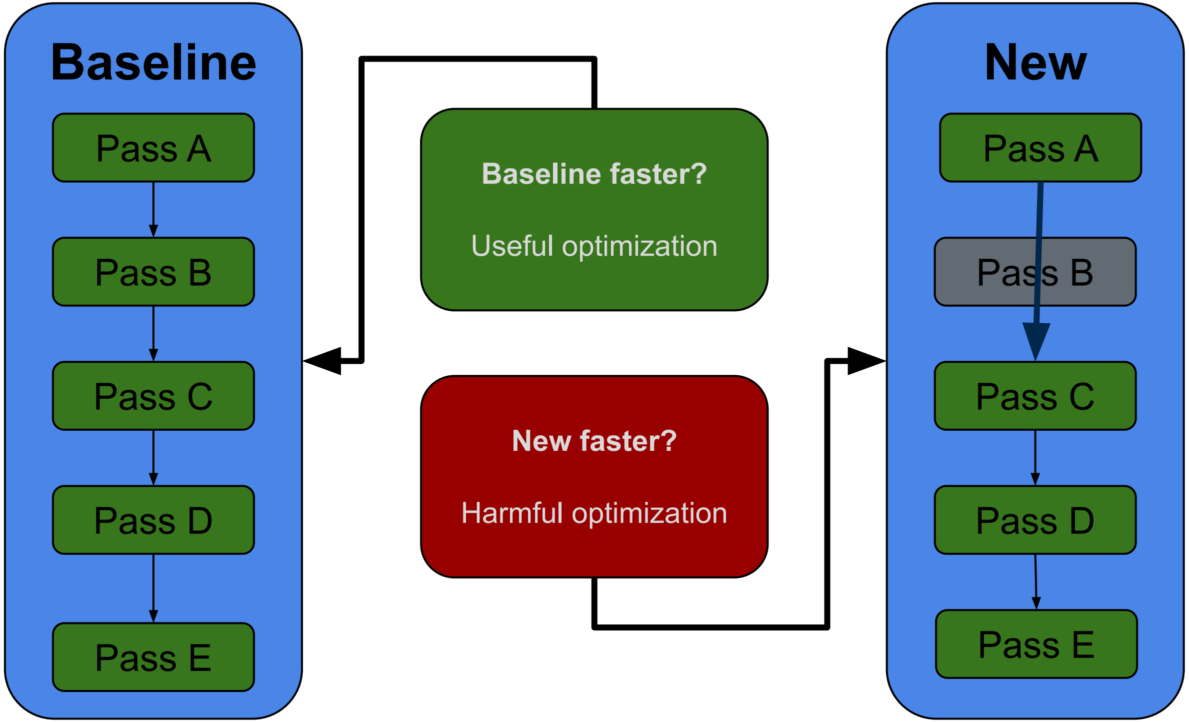
1. Establish Baseline time
2. Select a pass to test
3. **Deactivate Pass**
4. Recompile code
5. Profile new executable
6. Compare

Training: Reduction



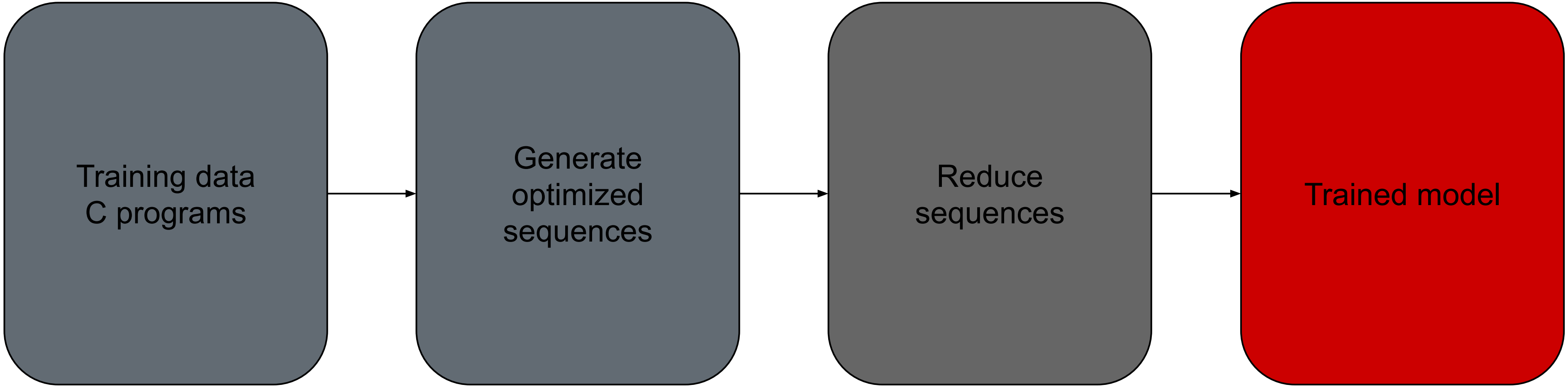
1. Establish Baseline time
2. Select a pass to test
3. Deactivate Pass
4. **Recompile code**
5. **Profile new executable**
6. Compare

Training: Reduction



1. Establish Baseline time
2. Select a pass to test
3. Deactivate Pass
4. Recompile code
5. Profile new executable
6. **Compare**

Training Completed

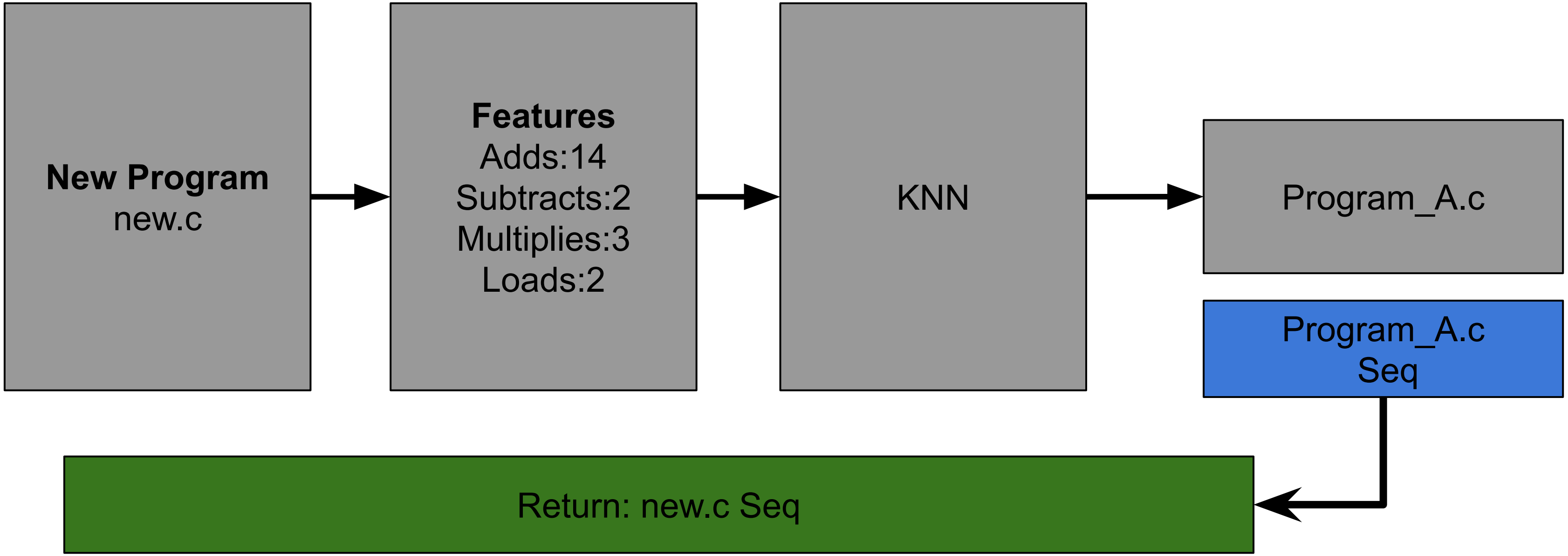


Step 4: Running the Model

- **Goal**: We want to apply auto-tuned passes on new programs
- KNN Model: Categories are programs
  - 1st nearest neighbor
- Distance Function: *"Cosine Similarity"*
  - Addresses "sparse data"

$$\text{Sim}(\text{p, pi}) = \frac{\sum_{w=1}^{m} Pw \times Piw}{\sqrt{\sum_{w=1}^{m} (Pw)^2} \times \sqrt{\sum_{w=1}^{m} (Piw)^2}}$$

Running the Model



Results - KNN

- Utilizing the KNN algorithm led to an average 21% enhancement in execution time.
- Utilizing the KNN algorithm plus the reduction led to an average 23% enhancement in execution time.

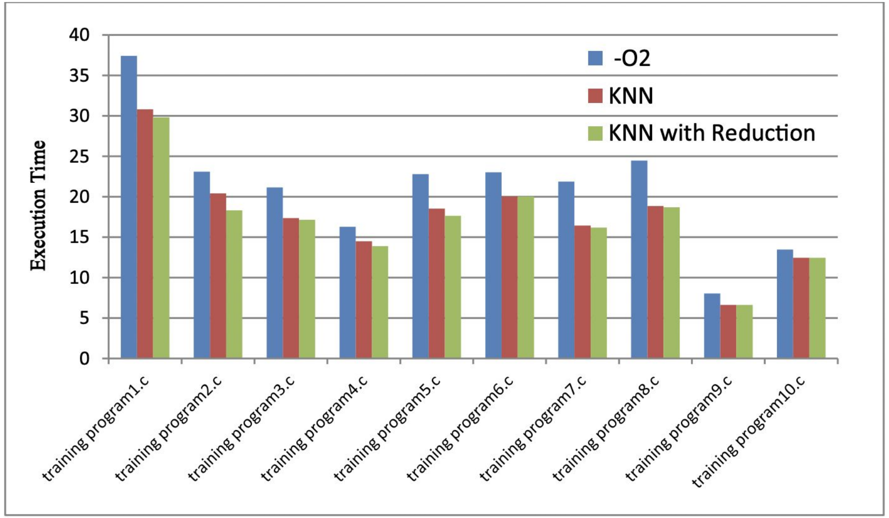


Fig. 3. The performance comparison between our approach and -O2 for the training set.

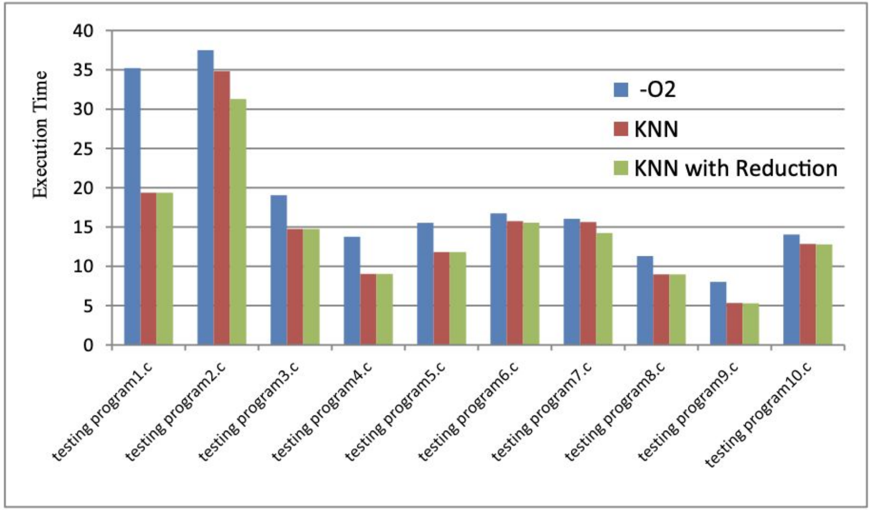


Fig. 4. The performance comparison between our approach and -O2 for the testing set.

Results - Number of Passes and Pass Order

- The combination of reduction algorithm and KNN resulted in an average 23% improvement in execution time
  - ~2% performance improvement attributed to reduction

Table 4. The optimization sequence before and after the reduction process in the testing set.

Programs	Optimization sequence before the reduction process	Number of passes	Optimization sequence after the reduction process	Number of passes
Testing program1.c	-prune-eh -early-cse -loop-rotate -loop-idiom -basicaa -basiccg -gvn -inline -jump-threading -loop-reduce -tailcallelim -instcombine -indvars -loop-deletion -licm	15	-prune-eh -early-cse -loop-rotate -loop-idiom -basicaa -basiccg -gvn -inline -jump-threading -loop-reduce -tailcallelim -instcombine -indvars -loop-deletion -licm	15
Testing program2.c	-early-cse -inline -prune-eh -loop-idiom -loop-rotate -basicaa -gvn -jump-threading -basiccg -loop-reduce -instcombine -tailcallelim -indvars -loop-deletion -licm	15	-early-cse -inline -prune-eh -loop-idiom -loop-rotate -basicaa -gvn -basiccg -instcombine -indvars -loop-deletion	11

Group Commentary

Pros

- Performance increase of 21% for KNN
- Performance increase of 23% for KNN plus reduction
- Interesting heuristic to find an optimized path on training data
- Finding optimization paths based on similar programs, useful for tailoring passes for specific categories of application

Group Commentary

Areas For Improvement

- Tested against O2 should have included O3
- Training greedily selects optimizations does not take into account interactions between optimizations
- Better test and train data
  - Only 10 data points tested
- Basic feature selection
  - Only instruction counts
- Don't maximize the classification abilities of KNN because of the small sample size (k=1)

Thank You!

Any Questions?