Machine Learning in Compilers

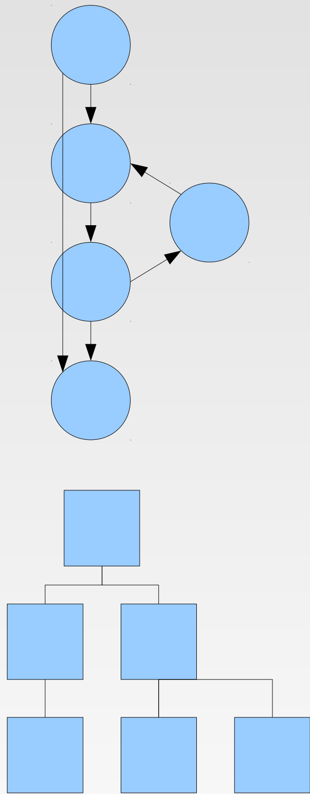
Institute for Computing Systems Architecture
University of Edinburgh, UK

Hugh Leather

Machine learning in compilers

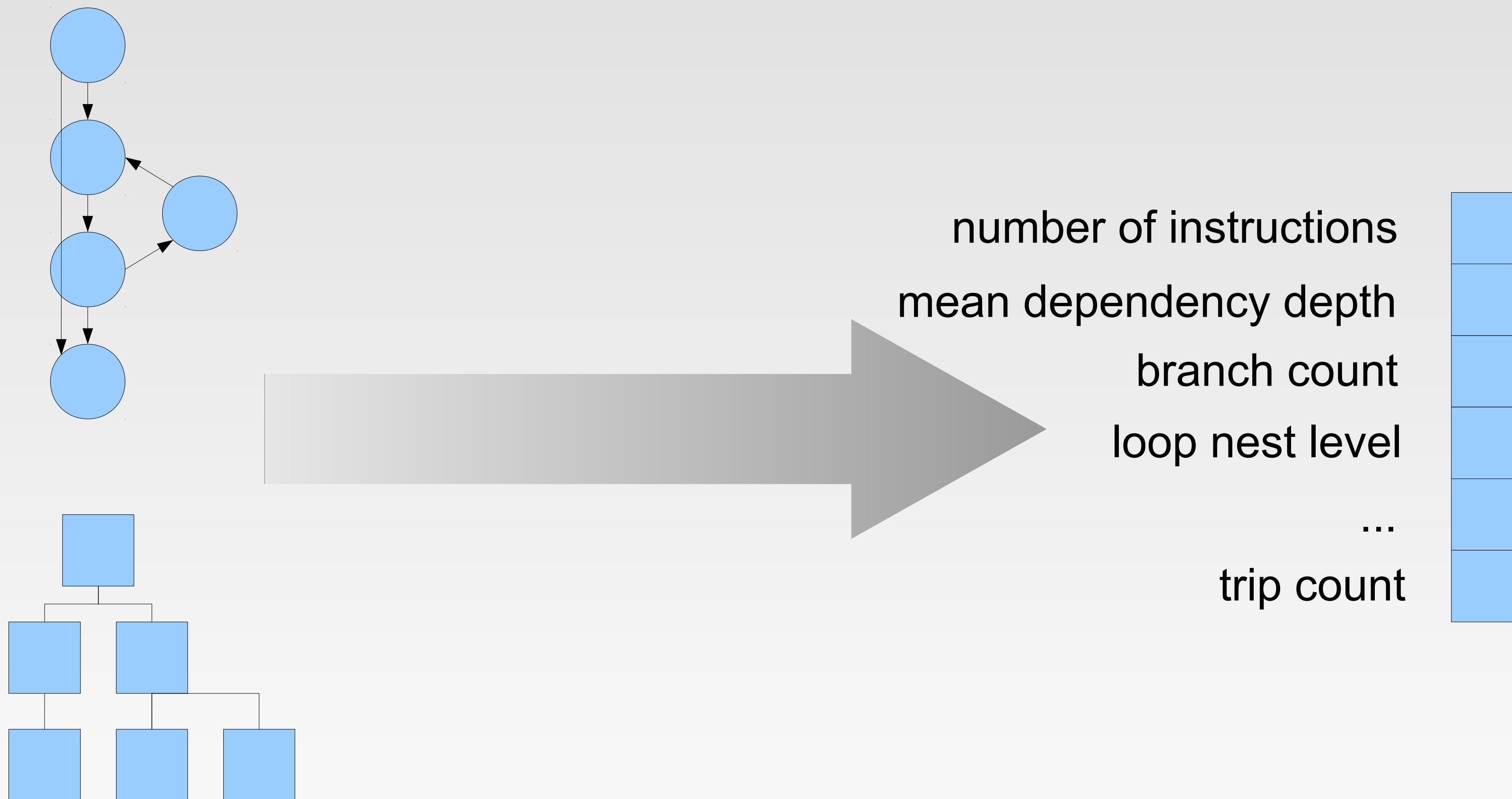
Start with compiler data structures

AST, RTL, SSA, CFG, DDG, etc.



Machine learning in compilers

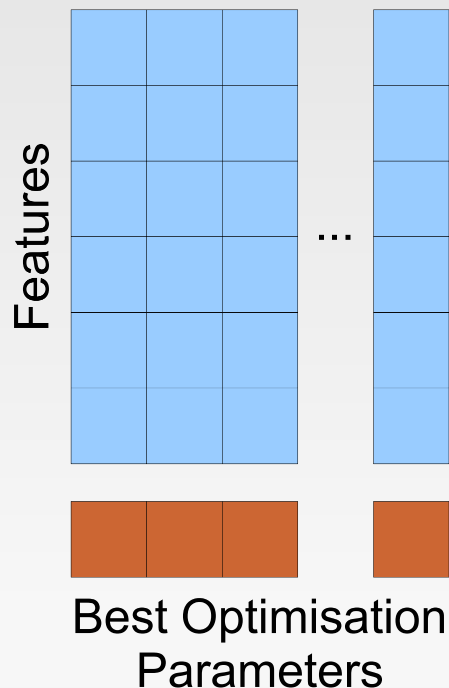
Human expert determines a mapping to a feature vector



Machine learning in compilers

Now collect many examples of programs, determining their feature values

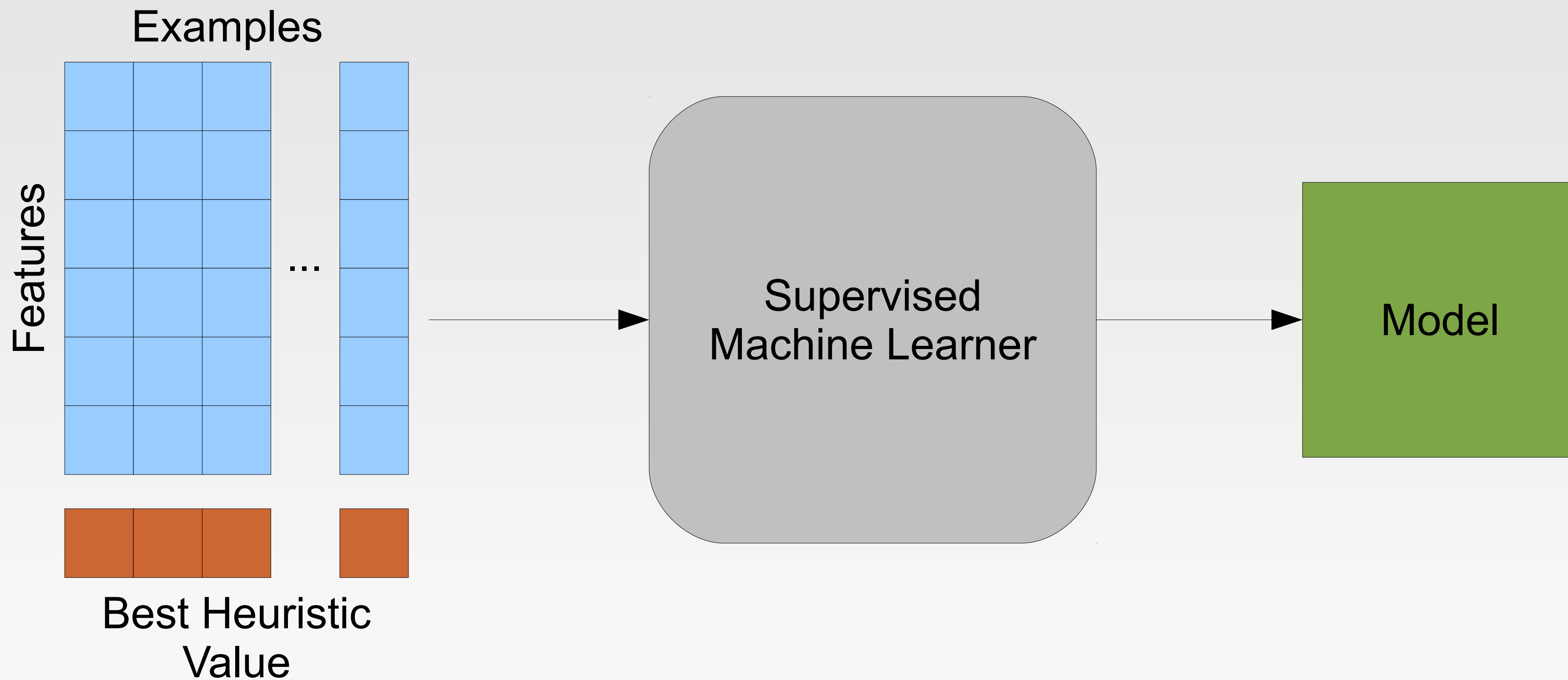
Execute the programs with different compilation strategies and find the best for each



Machine learning in compilers

Now give these examples to a machine learner

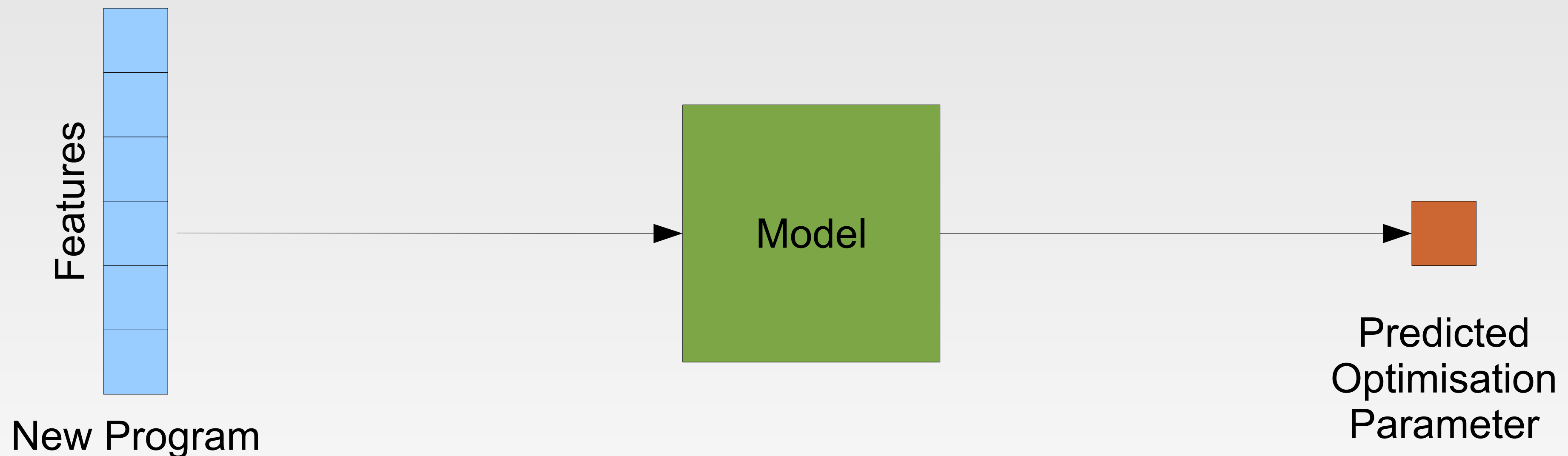
It learns a model



Machine learning in compilers

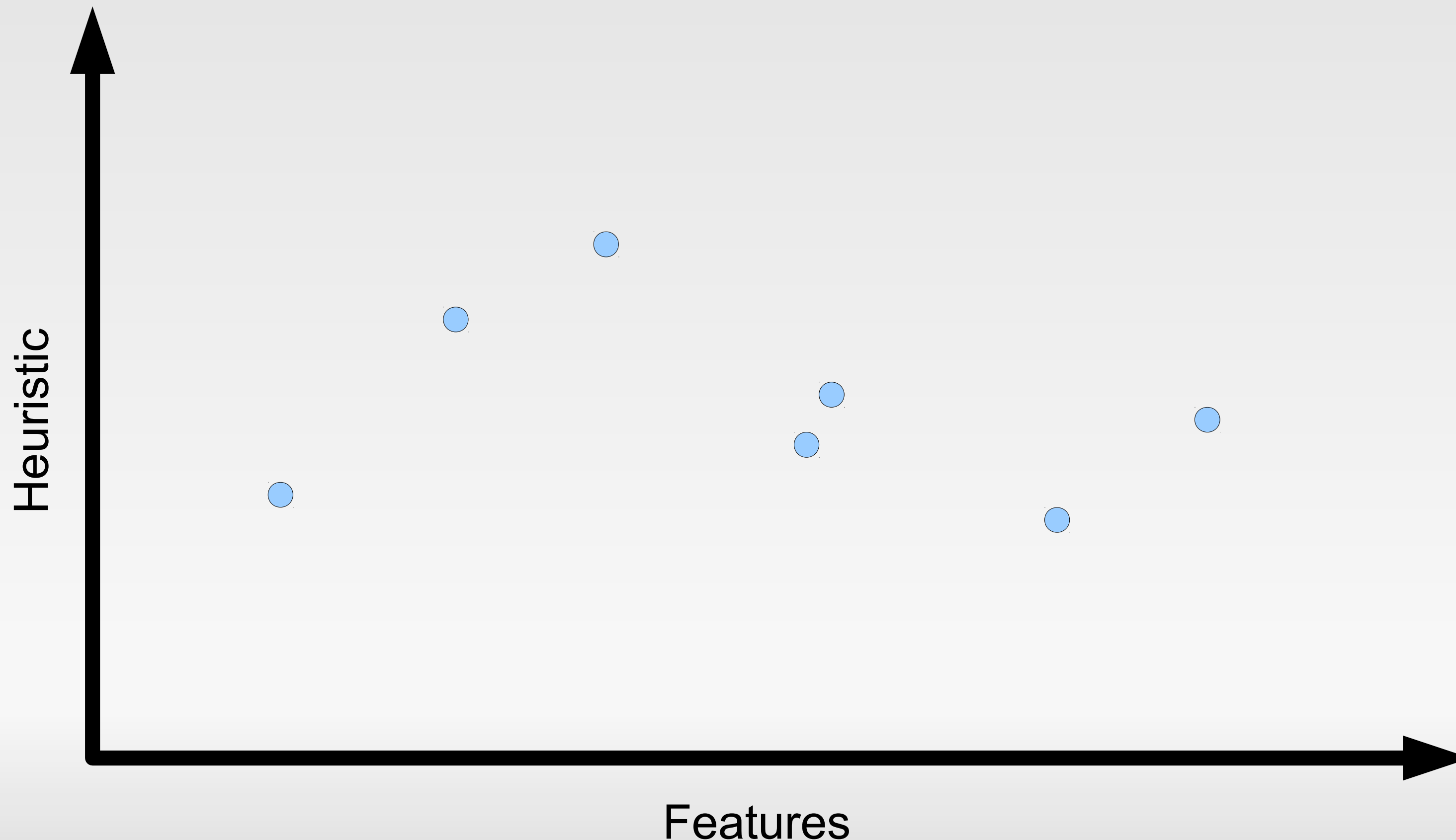
This model can then be used to predict the best compiler strategy from the features of a new program

Our heuristic is replaced



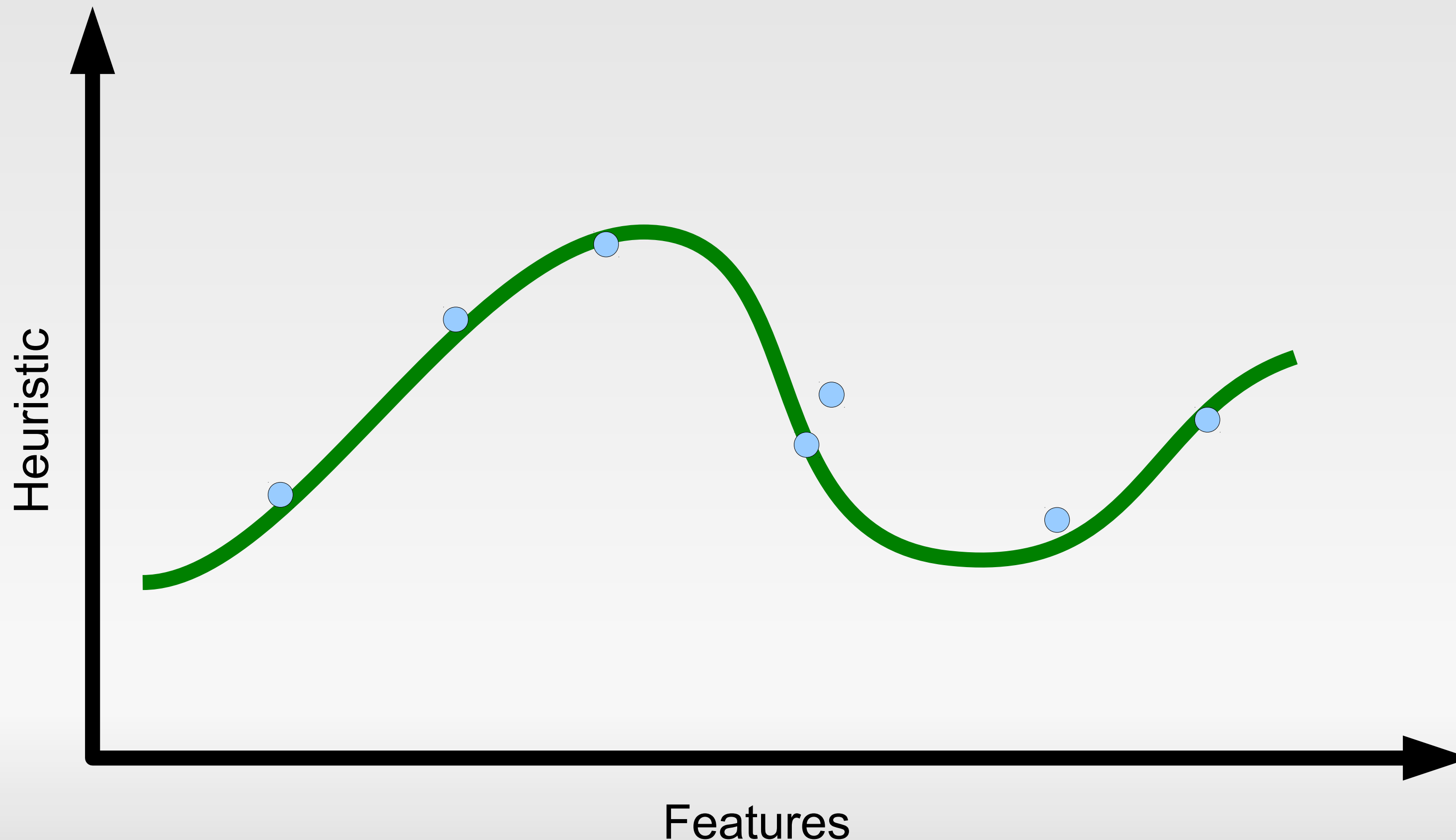
Machine learning in compilers

- A model is really just a way of fitting a curve to data



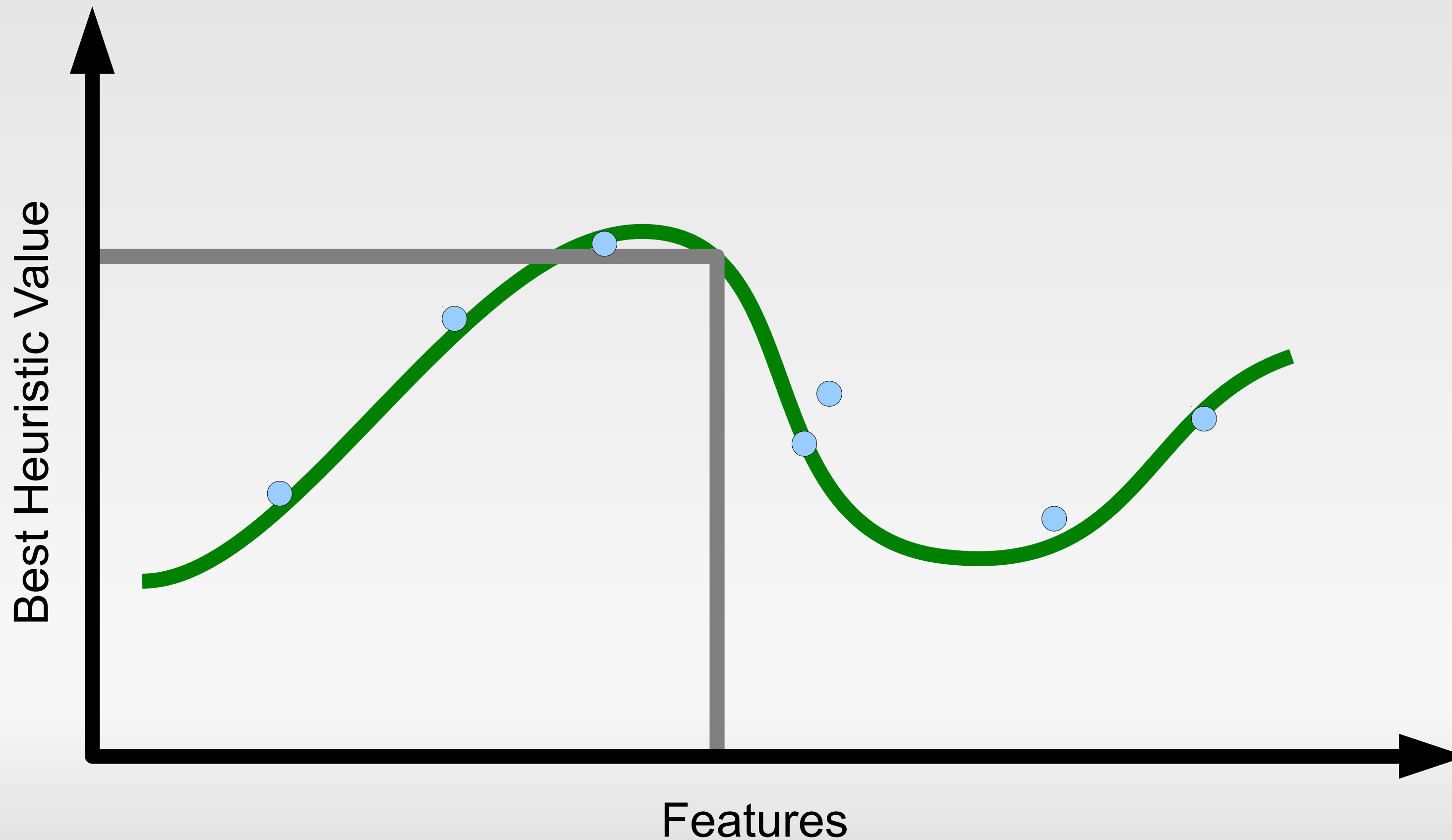
Machine learning in compilers

- A model is really just a way of fitting a curve to data



Machine learning in compilers

- Gives heuristic for unseen points

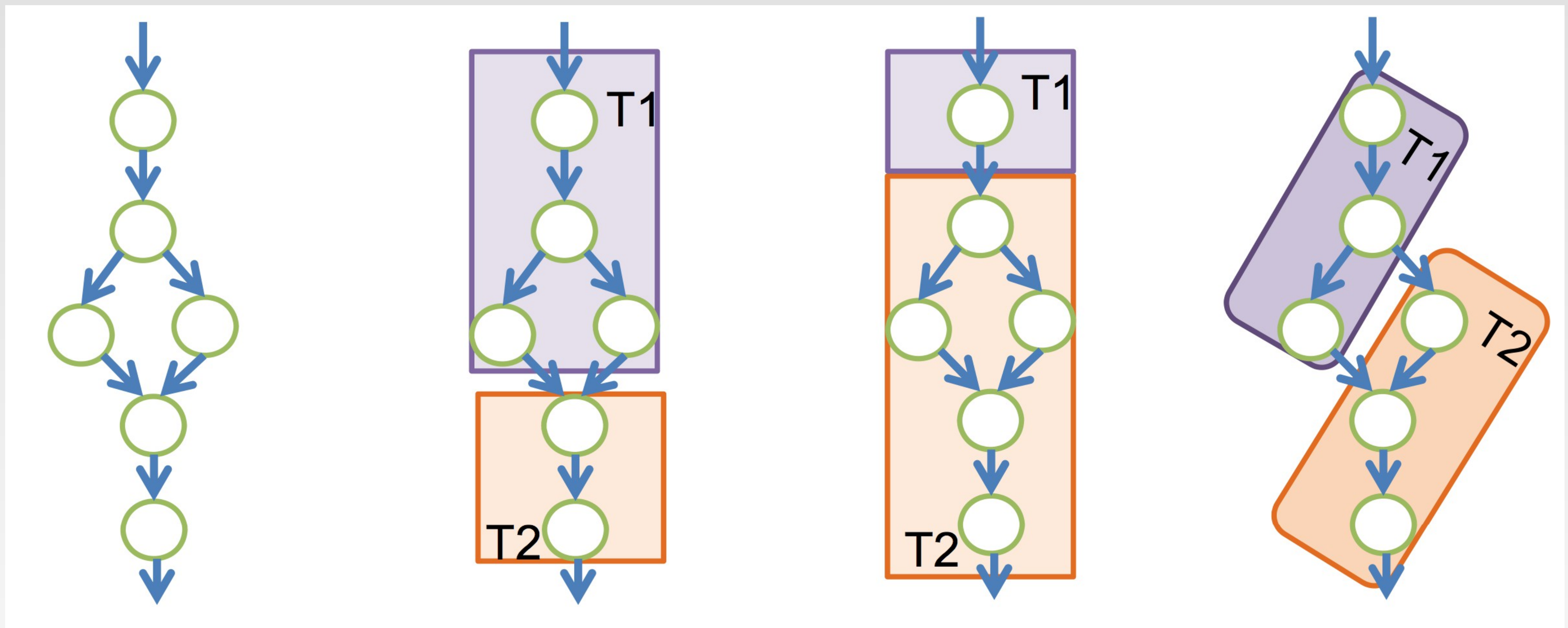


Example: Partitioning Stream Programs

Z. Wang, M. O'Boyle, Partitioning Streaming Parallelism for Multi-cores: A Machine Learning Based Approach, in **PACT 2010**

Partitioning Stream Programs

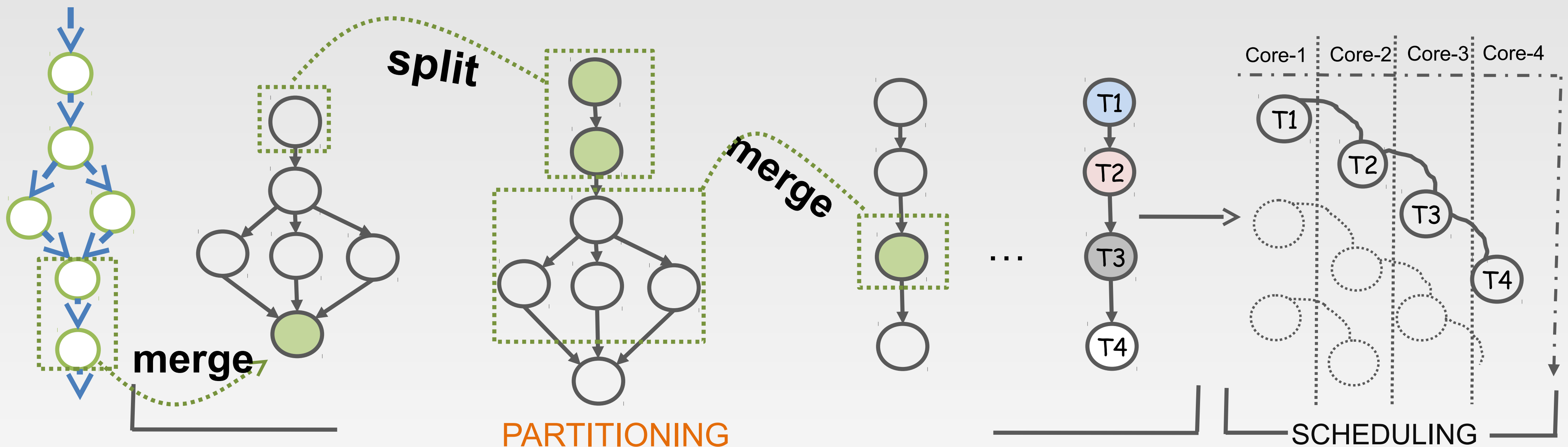
- Map the input program graph to threads
- Need to find a good one from many possible partitions



3 possible partitions on a 2-core machine

Generate A Partition

- Use a sequence of merging and splitting operations to generate a partition

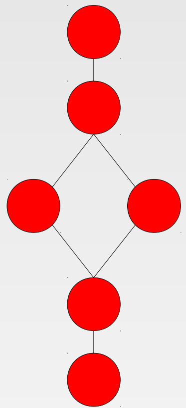


Compact graph representation.

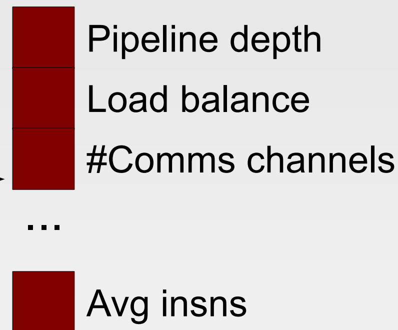
A Two Step Approach

1. Predict characteristics of the ideal partition

Input Graph



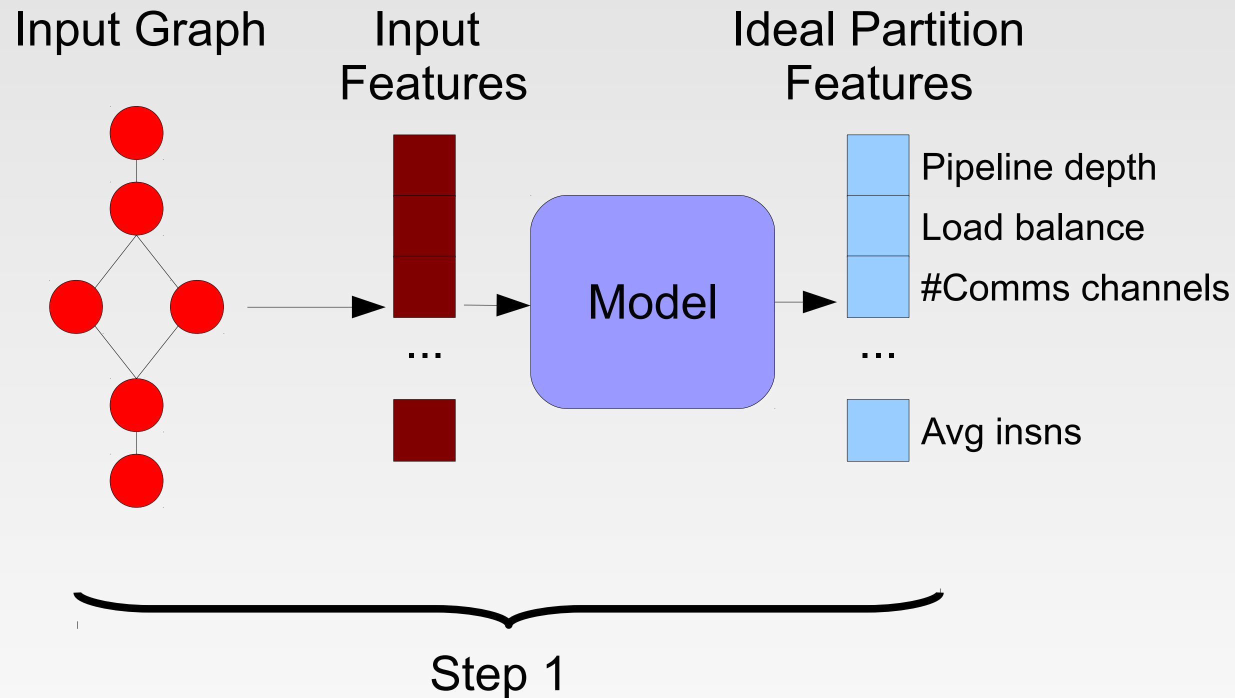
Input Features



We do NOT run any of the generated partition for searching

A Two Step Approach

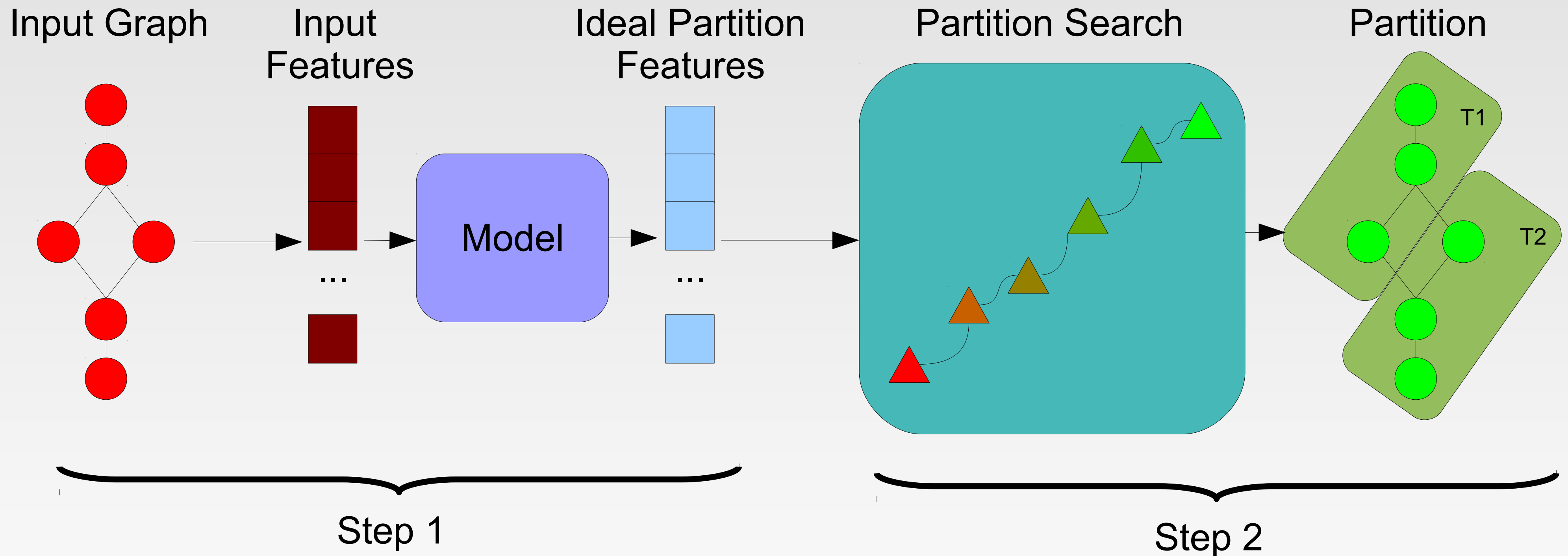
1. Predict characteristics of the ideal partition



We do NOT run any of the generated partition for searching

A Two Step Approach

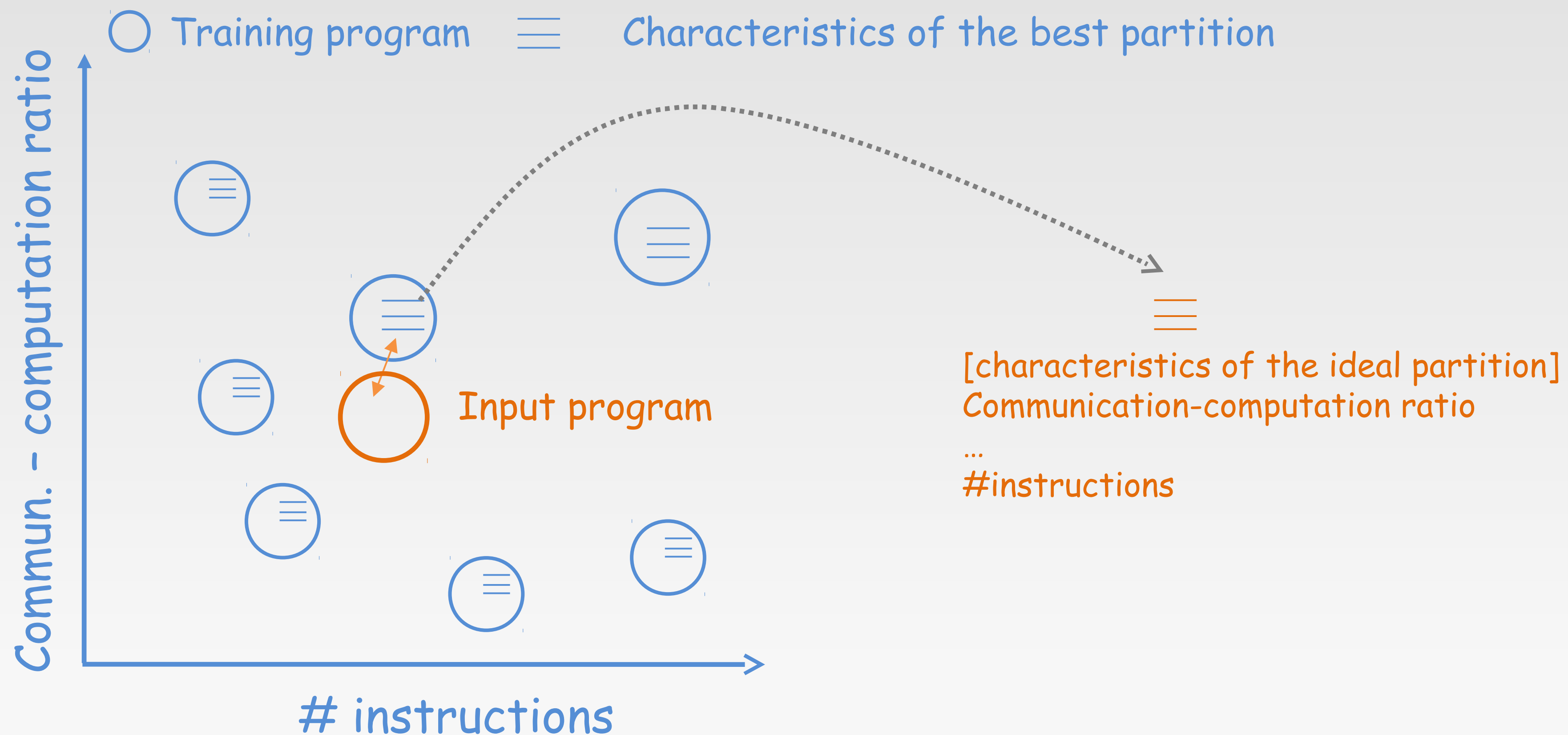
1. Predict characteristics of the ideal partition
2. Search for a partition with those characteristics



We do NOT run any of the generated partition for searching

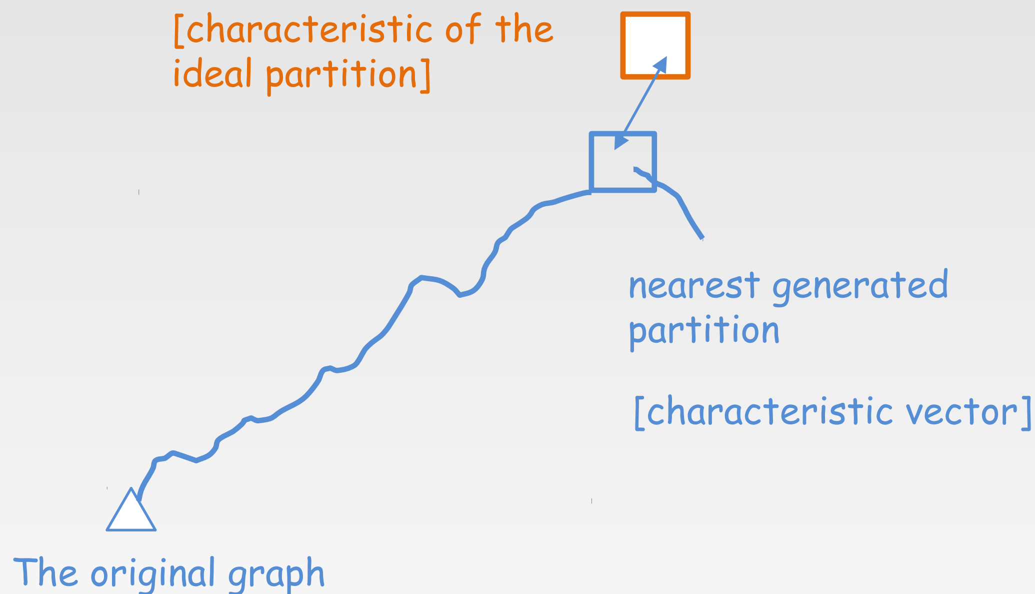
Step 1: Prediction

- Nearest neighbour algorithm to predict the characteristics of the ideal structure of the input program



Step 2: Search

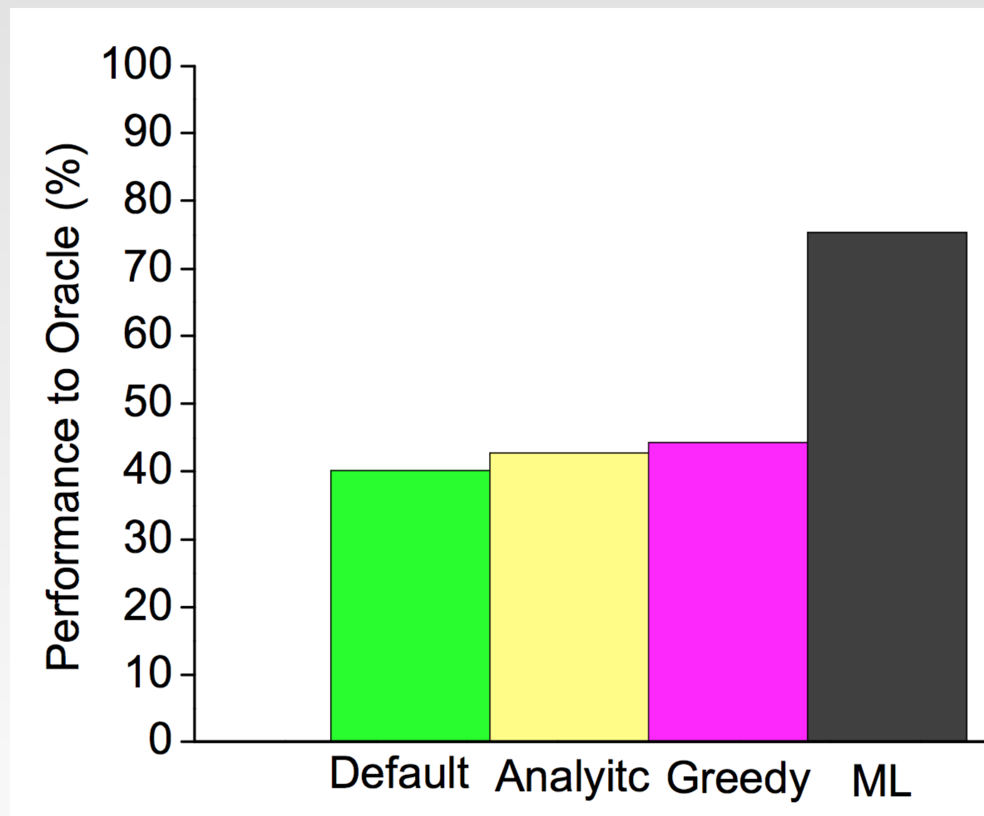
- Select a randomly generated partition whose structure is the most close to the predicted one



We do not run the program!

Results

- ML significantly outperforms state-of-the-art
- Not far from Oracle ("Best") performance



(Intel Xeon 4-Core)

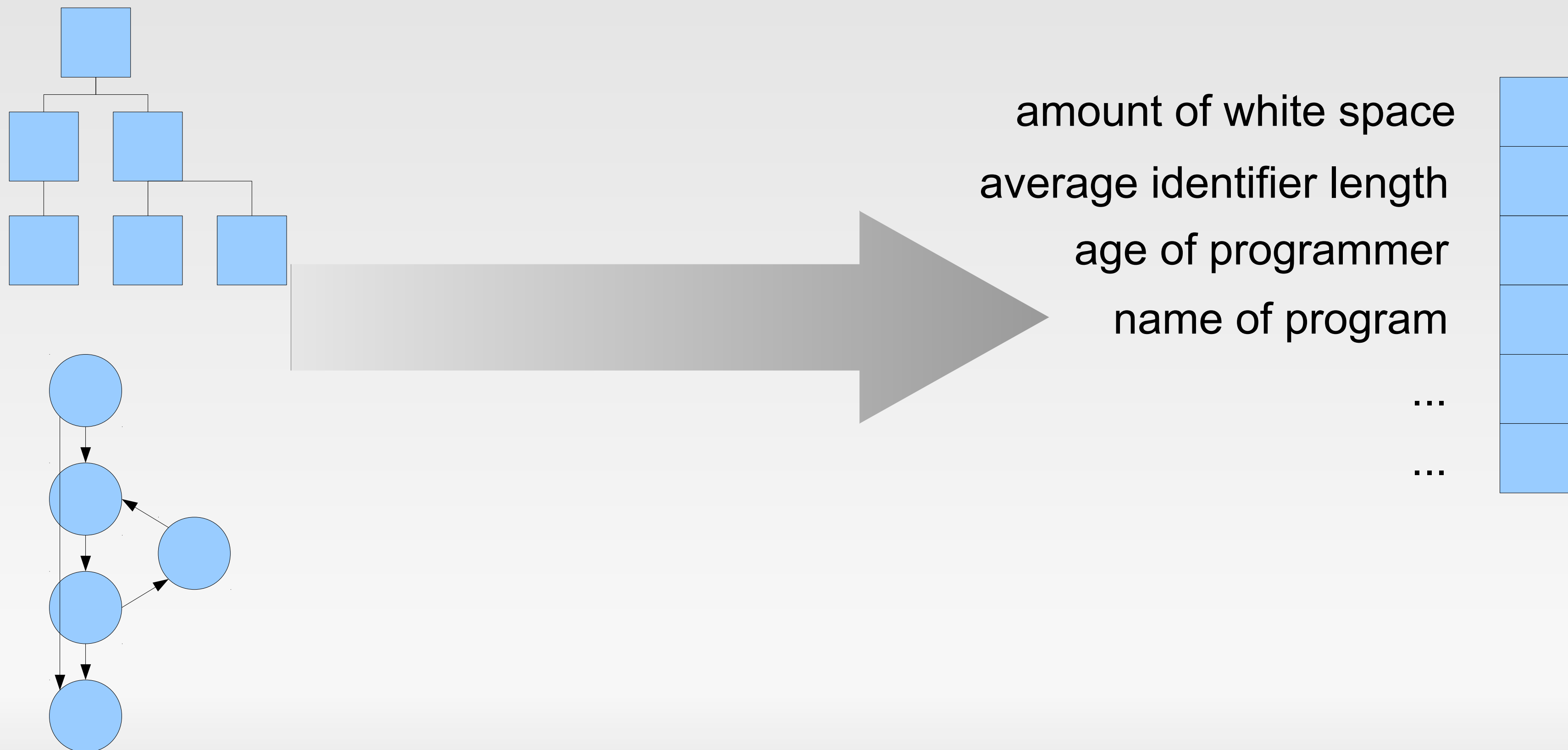
Automatic Feature Generation (Removing the human expert)

Choosing Features

- Problem
  - ML relies on good features
  - Subtle interaction between features and ML
  - Infinite number of features to choose from
- Solution
  - *Automatically search for good features!*

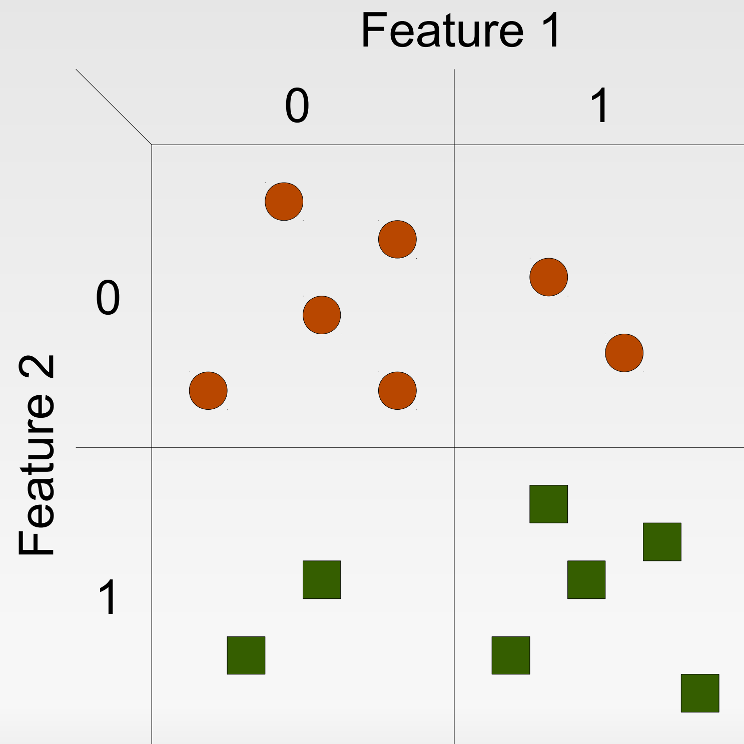
The Problem

- The expert must do a good job of projecting down to features



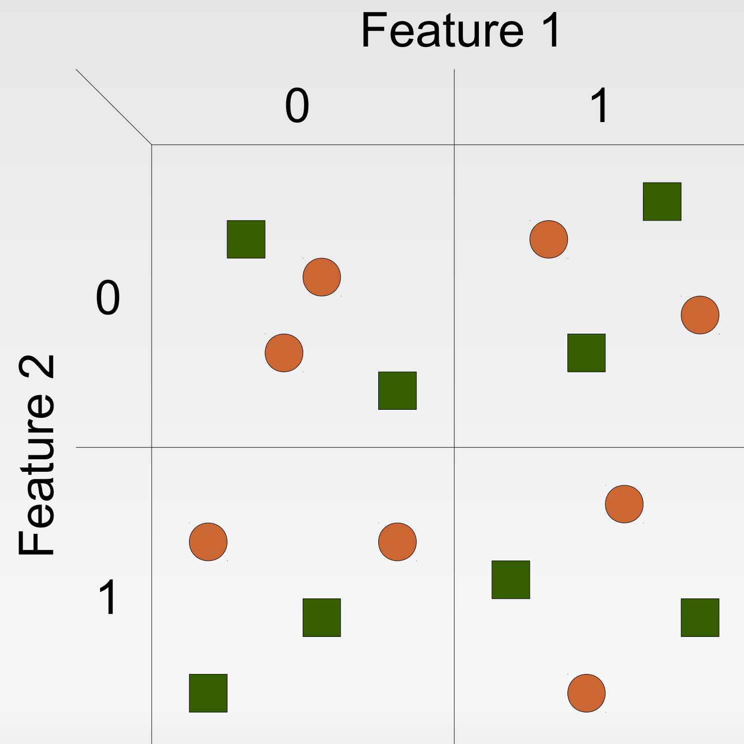
The Problem

- Machine learning works well when all examples associated with one feature value have the same type



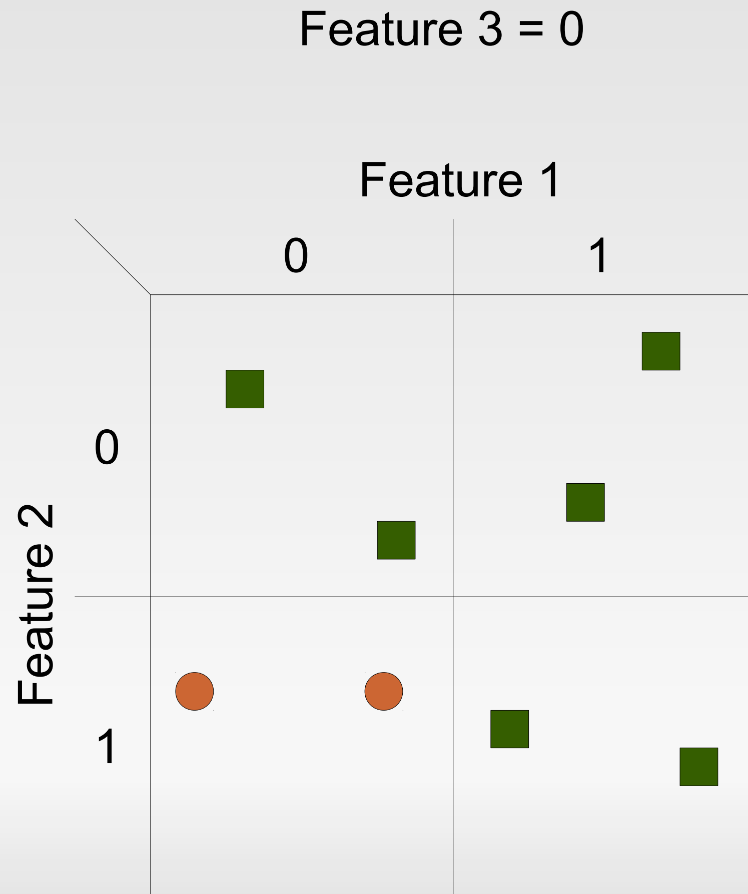
The Problem

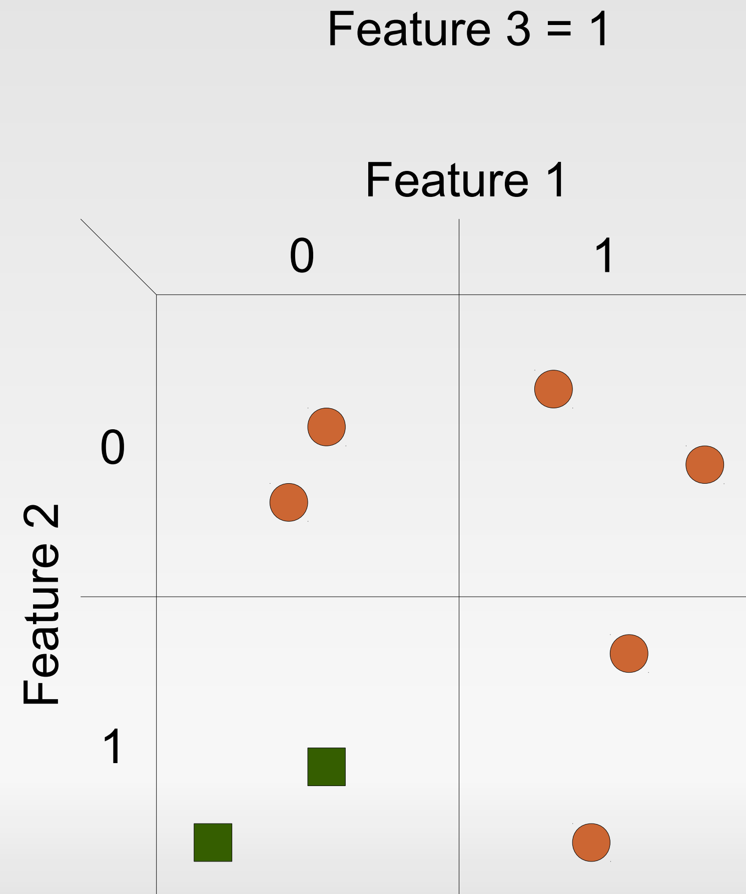
- Machine learning doesn't work if the features don't distinguish the examples



The Problem

- Better features might allow classification





The Problem

- There are much more subtle interactions between features and ML algorithm
  - Sometimes adding a feature makes things worse
  - A feature might be copies of existing features
- There is an infinite number of possible features

An example – Loop unrolling

- Set up
  - 57 benchmarks from MiBench, MediaBench and UTDSP
  - Found best unroll factor for each loop in [0-16]
  - Exhaustive evaluation to find oracle

An example – Loop unrolling

Original Loop

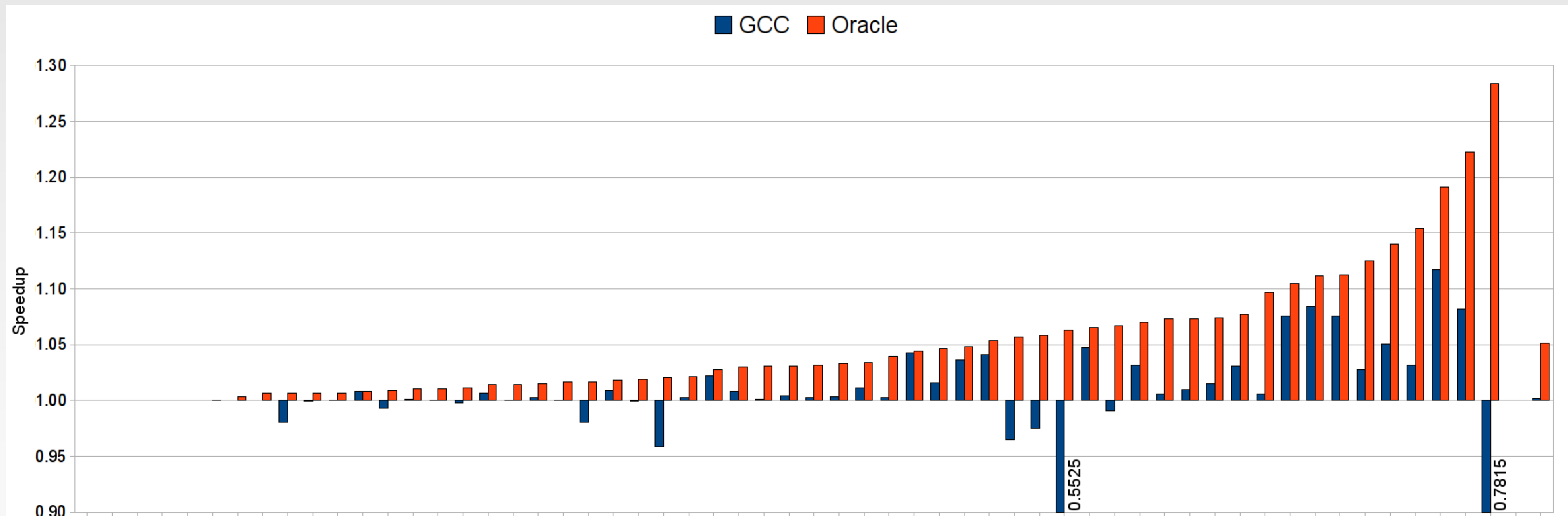
```
for( i = 0; i < n; i = i ++ ) {
    c[i] = a[i] * b[i];
}
```

Unrolled 5 times

```
for( i = 0; i < n; i = i + k ) {
    c[i+0] = a[i+0] * b[i+0];
    c[i+1] = a[i+1] * b[i+1];
    c[i+2] = a[i+2] * b[i+2];
    c[i+3] = a[i+3] * b[i+3];
    c[i+4] = a[i+4] * b[i+4];
    c[i+5] = a[i+5] * b[i+5];
}
```

GCC vs Oracle

- GCC gets 3% of maximum
- On average mostly not worth unrolling



State of the art features

- Lots of good work with hand-built features
  - Dubach, Cavazos, etc
- Stephenson was state of the art
  - Tackled loop unrolling heuristic
  - Spent some months designing features
  - Multiple iterations to get right

GCC vs Stephenson

- Gets 59% of maximum!
- Machine learning does well

GCC StateML

1.25
1.20
1.15
1.10
1.05
1.00
0.95

Speedup

0.5525
0.8038
0.7815

GCC vs Stephenson

	GCC	Stephenson
Heuristic	Months	
Features	-	Months
Training	-	Days
Learning	-	Seconds
Results	3%	59%

- To scale up, must reduce feature development time

A feature space for a motivating example

- Simple language the compiler accepts:
  - Variables, integers, '+', '*', parentheses
- Examples:
  - a = 10
  - b = 20
  - c = a * b + 12
  - d = a * ((b + c * c) * (2 + 3))

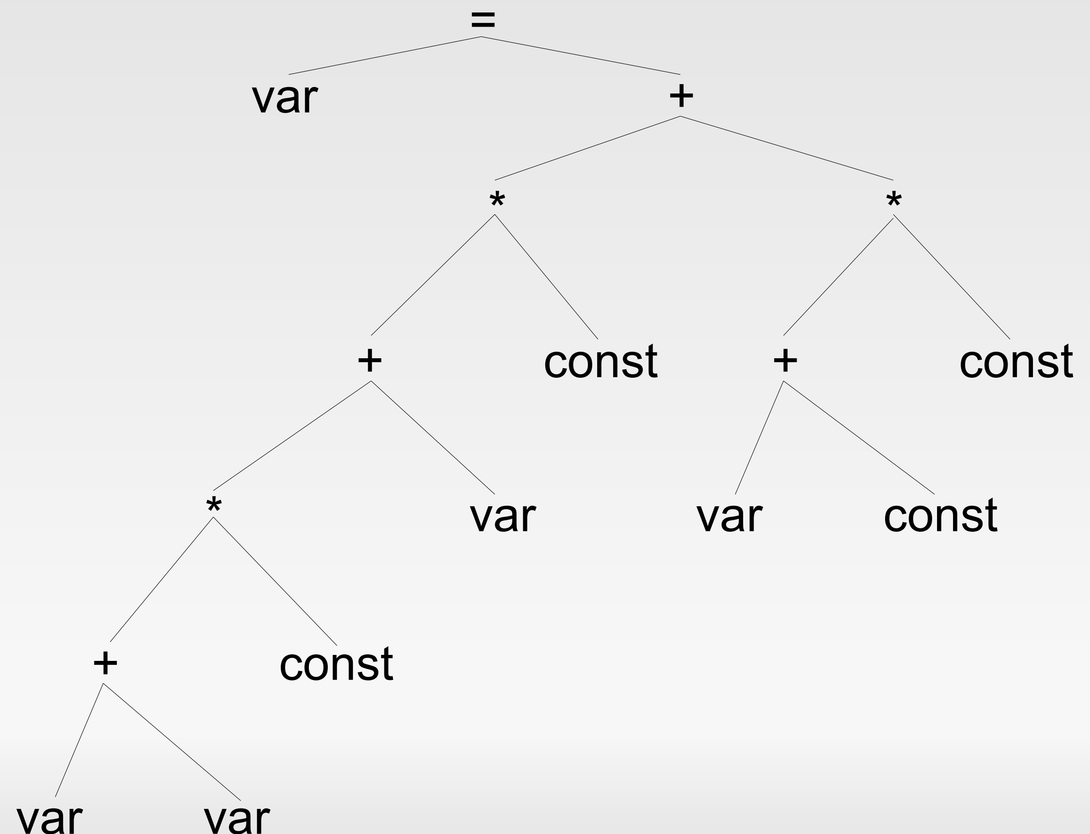
A feature space for a motivating example

- What type of features might we want?

a = ((b+c)*2 + d) * 9 + (b+2)*4

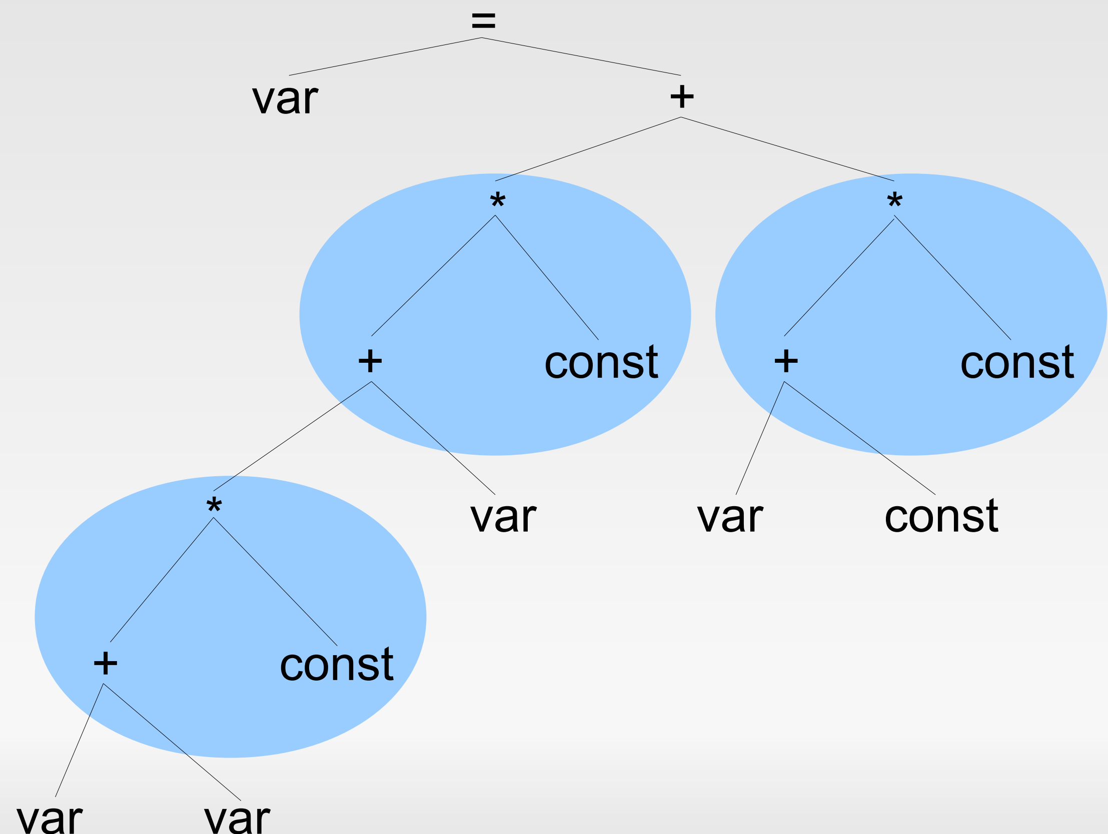
A feature space for a motivating example

- What type of features might we want?



A feature space for a motivating example

- What type of features might we want?

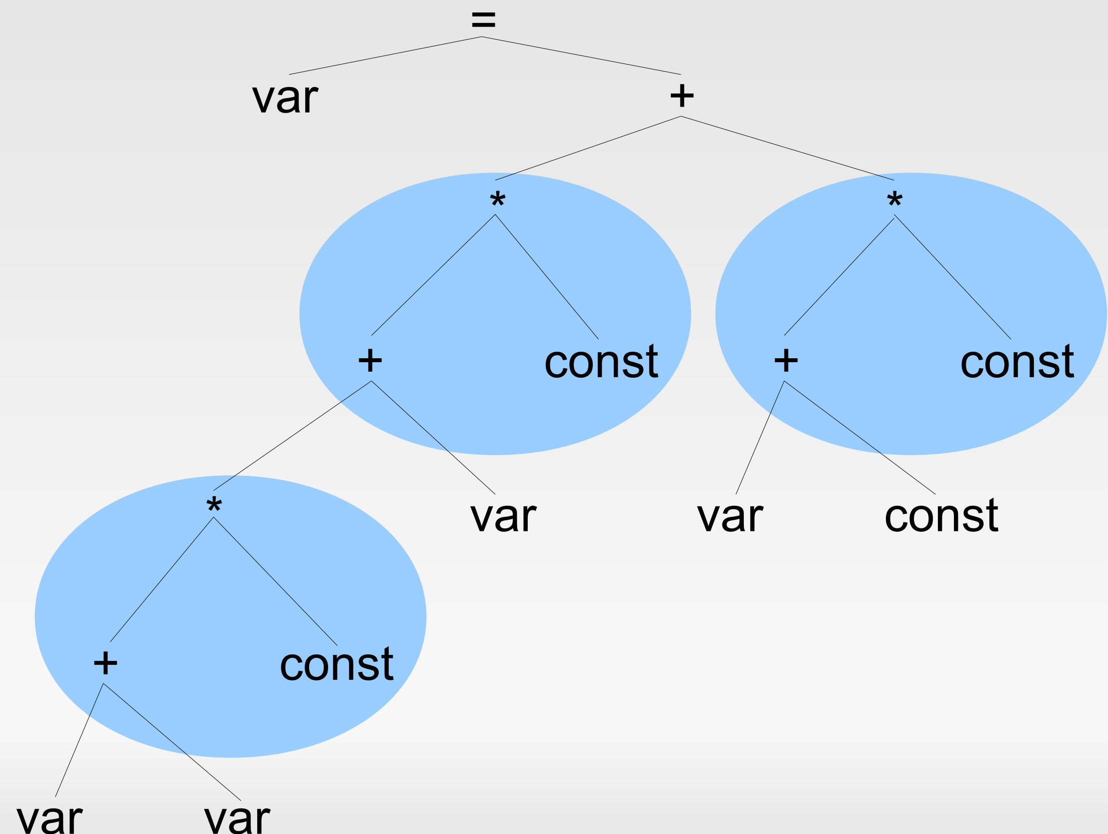


A feature space for a motivating example

- What type of features might we want?

```
count−nodes−matching(
    is−times &&
    left−child−matches(
        is−plus
    )&&
    right−child−matches(
        is−constant
    )
)
```

a = ((b+c)*2 + d) * 9 + (b+2)*4



Value = 3

A feature space for a motivating example

- Define a simple feature language:

```
<feature>   ::= ”count−nodes−matching(” <matches> ”)”
<matches>  ::= ”is−constant”
              |   ”is−variable”
              |   ”is−any−type”
              |   ( ”is−plus” | ”is−times” )
                  ( ”&& left−child−matches(” <matches> ”)” ) ?
                  ( ”&& right−child−matches(” <matches> ”)” ) ?
```

- GCC grammar is huge >160kb
- Genetic search for features that improve machine learning prediction

Generate a feature from a grammar

- Now generate sentences from the grammar to give features
- Start with the root non-terminal

Grammar			Sentence
<A>	::=	<A><A><A>	A
	|	"b"	

Generate a feature from a grammar

- Now generate sentences from the grammar to give features
- Choose randomly among productions and replace

Grammar		Sentence
<A> ::=	<A><A><A>	AAA
|	“b”	

Generate a feature from a grammar

- Now generate sentences from the grammar to give features
- Repeat for each non-terminal still in the sentence

Grammar		Sentence
<A> ::=	<A><A><A>	bAAAb
|	“b”	

Generate a feature from a grammar

- Now generate sentences from the grammar to give features
- Continue until there are no more non-terminals

Grammar
<A> ::= <A><A><A>
| "b"

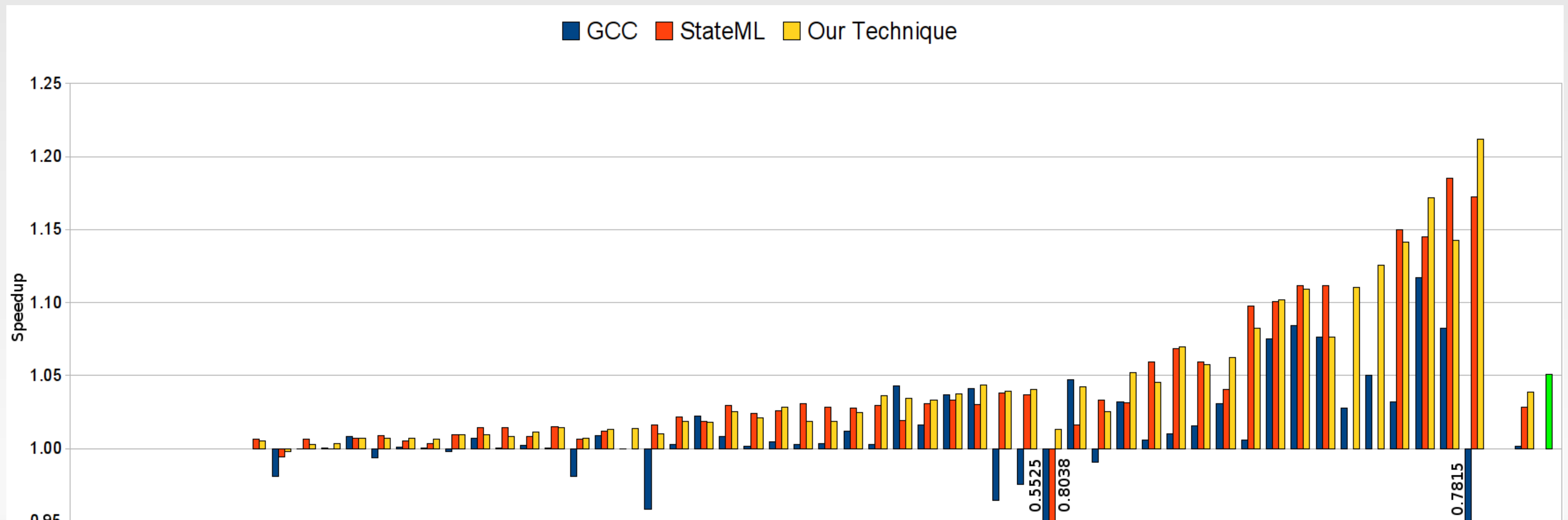
Sentence
bbbbb

Genetic search over features

- Search space is parse trees of features
- Genetic programming searches over feature parse trees
- Features which help machine learning are better

Results

- GCC 3%   Stephenson 59%   Ours 75%
- Automated features outperform human ones



Results

- Top Features Found

39%
  - get-attr(@num-iter)

Results

- Top Features Found

39% - get-attr(@num-iter)

14% - count(filter(//*, !(is-type(wide-int) || (is-type(float extend) &&[(is-type(reg)]/count(filter(//*,is-type(int))))) || is-type(union type))))

Results

- Top Features Found

39% ▪ get-attr(@num-iter)

14% ▪ count(filter(//*, !(is-type(wide-int) || (is-type(float extend) &&[(is-type(reg)]/count(filter(//*,is-type(int))))) || is-type(union type))))

8% ▪ count(filter(/*, (is-type(basic-block) && (

```
    !@loop-depth==2 ||
    (0.0 > (
        (count(filter(//*, is-type(var decl))) -
        (count(filter(//*, (is-type(xor) && @mode==HI))) +
        sum(
            filter(/*, (is-type(call insn) && has-attr(@unchanging))),
            count(filter(//*, is-type(real type)))))) /
            count(filter(/*, is-type(code label))))))))
```

GCC vs Stephenson vs Ours

	GCC	Stephenson	Ours
Heuristic	Months	-	**-**
Features	-	Months	**-**
Training	-	Days	**Days**
Learning	-	Seconds	**Hours**
Results	3%	59%	**75%**

Conclusion

- Analytic approaches no longer working
- Iterative compilation
  - Empirical and good but too slow
- Machine learning here to stay
  - Outperforming human heuristics
  - Very fast development time
- Now used for many things
  - Multi-core, GPGPU, Mobile, JIT, SQL, etc.