How to select the best graph representation for a given task

Alexander Slesarenko
Alexander Filippov
Alexey Romanov
Yuri Zotov

GraphHPC'15, March 5, Moscow

www.huawei.com

Huawei Technologies Co., Ltd.

HUAWEI

Let’s consider the task

$$\overset{M}{\begin{pmatrix} 1.0 & 0 & 2.0 & 0 \\ 3.0 & 4.0 & 5.0 & 0 \\ 0 & 0 & 0 & 6.0 \end{pmatrix}} * \overset{V}{\begin{pmatrix} 1.0 \\ 2.0 \\ 3.0 \\ 4.0 \end{pmatrix}} = \overset{R}{\begin{pmatrix} 7.0 \\ 26 \\ 24 \end{pmatrix}}$$

Abstract data types

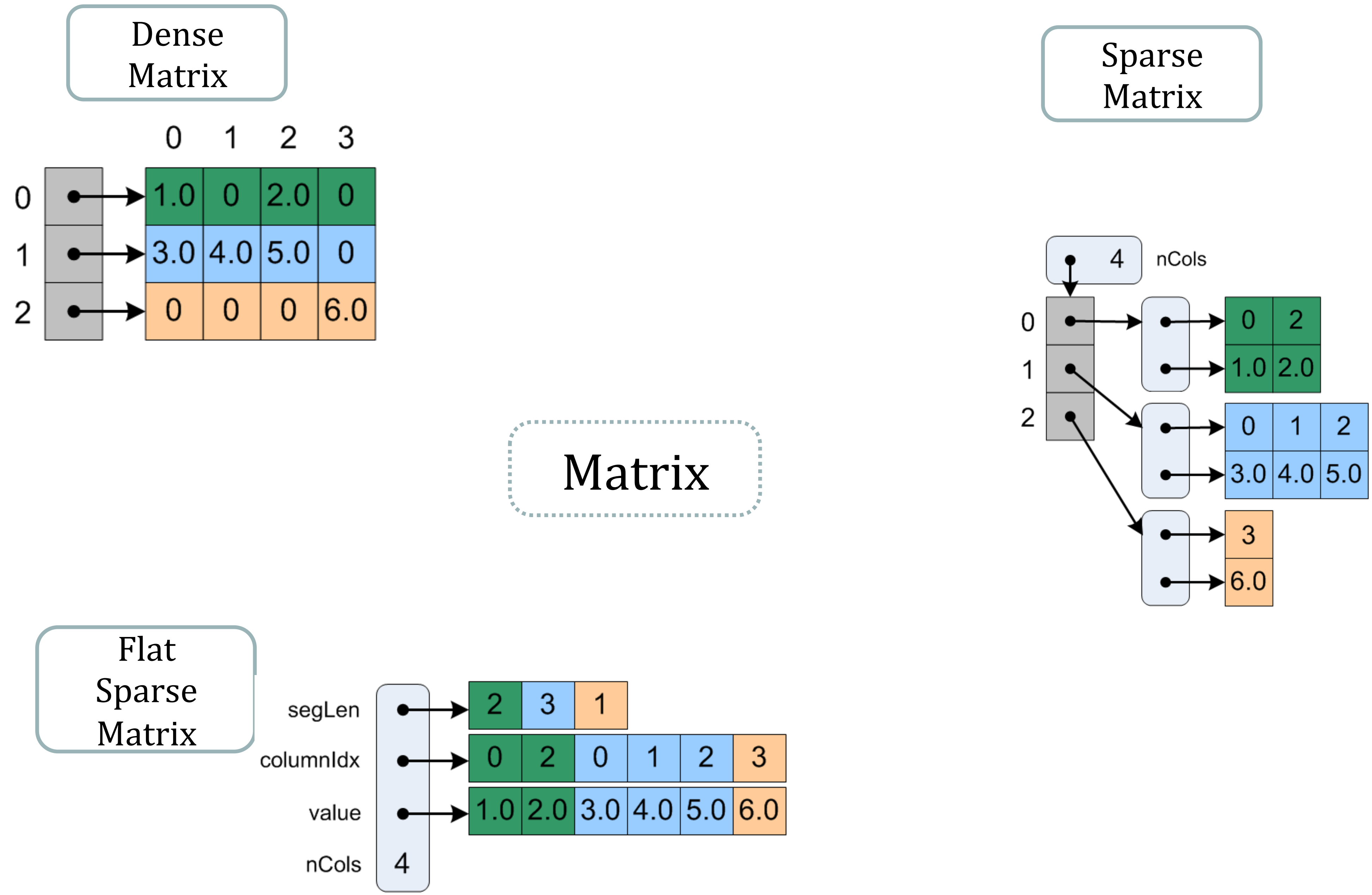
```
def mvm(m: Matr, vec: Vec): Vec = {
  Vec(m.rows.map(r => r.dot(vec)))
}
```

We can construct a new vector from an array of values

We can retrieve rows from the matrix as an array of vectors

when we map an Array we create a new Array

The problem: which representation is better?



Why it is a problem?

Consider matrix 10^4 x 10^4

$\mathbf{S_m}$ - matrix sparseness (% of zeros)

$\mathbf{S_v}$ - vector sparseness

S_m	S_v	**dmdv**	**dmsv**	**smdv**	**smsv**
0%	0%	309	354	366	760
10%	10%	311	323	332	1002
50%	50%	310	202	187	924
90%	90%	307	104	42	172
99%	99%	307	18	8	18
0%	50%	308	198	373	1134
50%	0%	310	359	187	986
10%	90%	311	118	335	497
90%	10%	311	323	42	345

Execution time in milliseconds

How do you know this is a bad choice?

The solution

```scala
def mvm(m: Matr, vec: Vec): Vec = {
  Vec(m.rows.map(r => r.dot(vec)))
}
```

$$
\underset{M}{\begin{pmatrix} 1.0 & 0 & 2.0 & 0 \\ 3.0 & 4.0 & 5.0 & 0 \\ 0 & 0 & 0 & 6.0 \end{pmatrix}} * \underset{V}{\begin{pmatrix} 1.0 \\ 2.0 \\ 3.0 \\ 4.0 \end{pmatrix}} = \underset{R}{\begin{pmatrix} 7.0 \\ 26 \\ 24 \end{pmatrix}}
$$

Automatically: generate, run and measure performance

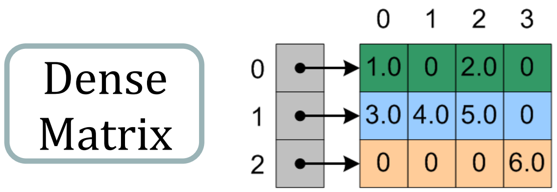
```scala
def dmdvm(m: Array[Array[Double]], v: Array[Double]): Array[Double] =
    m.map(row => sum(row |*| v))

def dmsvm(m: Array[Array[Double]], v: (Array[Int], Array[Double], Int)): Array[Double] = {
  val (indices, values, _) = v
  m.map(row => sum(row(indices) |*| values))}

def smsvm(m: Array[ (Array[Int],Array[Double],Int) ],
          v: (Array[Int],Array[Double],Int)): Array[Double] = {
  val (indices, values, _) = v
  m.map((is,(vs,_)) => dotProductSV(is, vs, indices, values))}

def smdvm(m: Array[(Array[Int], (Array[Double], Int))], v: Array[Double]): Array[Double] =
  m.map(r => {
    val (indices, values, _) = r
    sum(values |*| v(indices))
  })
```

Implementation-specific primitives:

- map
- sum
- |*|
- dotProductSV

Two stages of compilation

```
def mvm(m: Matr, vec: Vec): Vec =
  Vec(m.rows.map(r => r.dot(vec)))
```



Isomorphic Specialization into Core language

```
def dmdvm(m: Array[Array[Double]], v: Array[Double]): Array[Double] =
  m.map(row => sum(row |*| v))
```

Core language compilation with loop fusion, deforestation etc.

```
def dmdv(m: Array[Array[Double]], v: Array[Double]): Array[Double] = {
    val nRows = m.length
    var res = new Array[Double](nRows)
    for (i <- 0 until nRows) {
        val row = m(i)
        val nCols = row.length
        var sum: Double = 0
        for (j <- 0 until nCols) {
            sum += row(j) * v(j)
        }
        res(i) = sum
    }
    res
}
```

First-class Isomorphic Specialization by Staged Evaluation

Alexander Slesarenko
Shannon Laboratory, Huawei Technologies, Moscow, Russia
alexander.slesarenko@huawei.com

Alexander Filippov
Shannon Laboratory, Huawei Technologies, Moscow, Russia
filippov.alexander@huawei.com

Alexey Romanov
Shannon Laboratory, Huawei Technologies, Moscow, Russia
alexey.romanov@huawei.com

Abstract

The state of the art approach for reducing complexity in software development is to use abstraction mechanisms of programming languages such as modules, types, higher-order functions etc. and develop high-level frameworks and domain-specific abstractions. Abstraction mechanisms, however, along with simplicity, introduce also execution overhead and often lead to significant performance degradation. Avoiding abstractions in favor of performance, on the other hand, increases code complexity and cost of maintenance.

We develop a systematic approach and formalized framework for implementing software components with a first-class specialization capability. We show how to extend a higher-order functional language with abstraction mechanisms carefully designed to provide automatic and guaranteed elimination of abstraction overhead.

We propose *staged evaluation* as a new method of program staging and show how it can be implemented as zipper-based traversal of program terms where one-hole contexts are generically constructed from the abstract syntax of the language.

We show how generic programming techniques together with staged evaluation lead to a very simple yet powerful method of *isomorphic specialization* which utilizes first-class definitions of isomorphisms between data types to provide guarantee of abstraction elimination.

straction mechanisms (such as module systems, classes, interfaces, etc.). These mechanisms are often used to create domain-specific languages (DSLs) which allow a higher level of abstraction for programs in a given domain (e.g. Spark [33] can be considered as a DSL for distributed programming).

However, these mechanisms generally also introduce execution overhead (often called *abstraction regret* [4, 21] or *abstraction penalty*) and the trade-off between abstraction and performance is often difficult.

Modern advances in compilation techniques, such as just-in-time compilation and whole program optimization generally can't eliminate the overhead completely and don't scale well with the size of the program.

A recent trend is development of DSL-centric frameworks where abstractions can be introduced and software can be built without abstraction penalty [5, 23, 24], though it may require development of special tools [3].

In such frameworks, DSL compilers allow mapping of problem-specific abstractions directly to low-level architecture-specific programming models such as [12, 20]. However, the development of DSLs is difficult by itself, and adding a compilation stage considerably increases this difficulty.

While compiling DSLs is a promising approach, we believe that

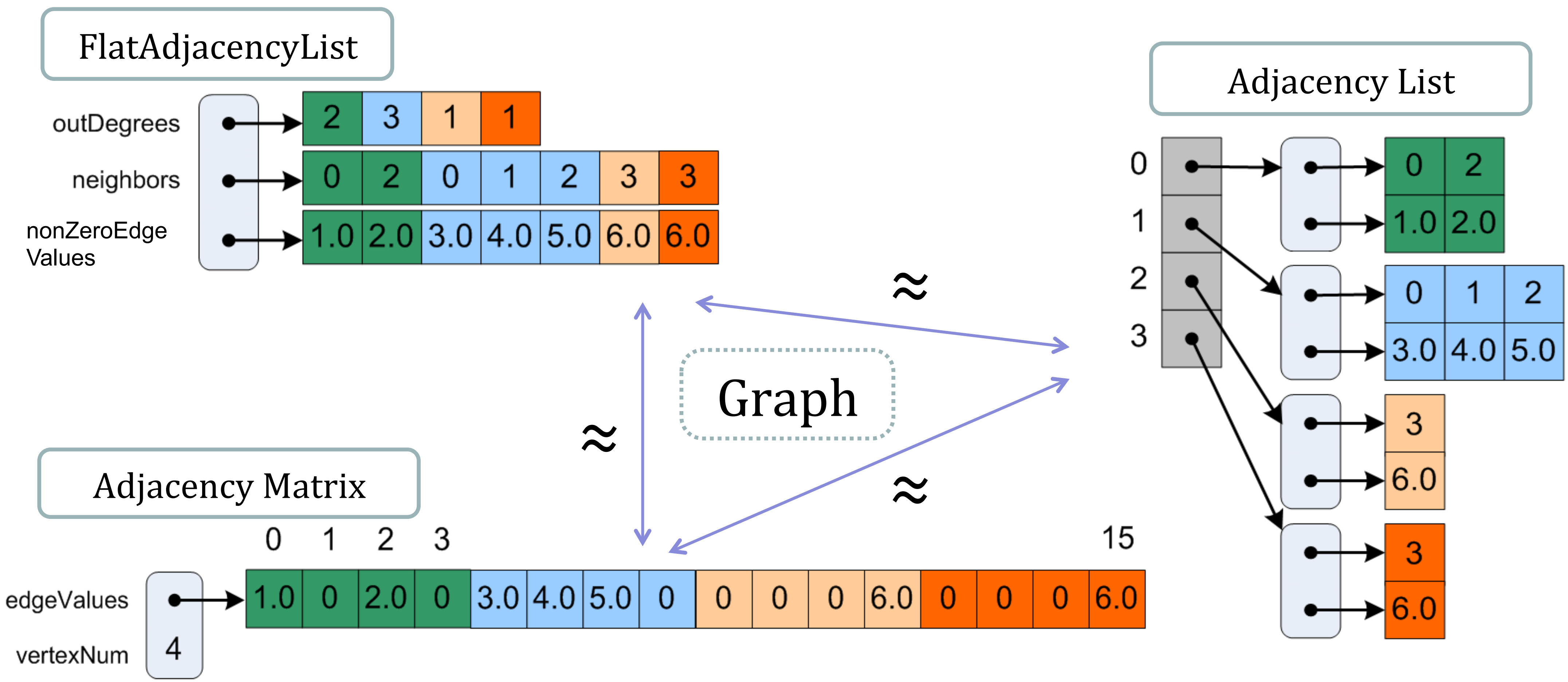
MST

```
def MST_prim(g: Graph, startFront: Front): Coll[Int] =
{
  def stopCondition(front: Front, tree: Coll[Int]) =
       (g.outEdgesOf(front).length === 0)

  def doStep(front: Front, tree: Int) = {
    val outEdges = g.outEdgesOf(front)
    val (_, (minFrom, minTo)) = outEdges
         .map(e => (e.value, (e.fromId, e.toId))})
         .reduce(MinWeightMonoid)
    (front.append(minTo), tree.update(minTo, minFrom))
  }

  val initTree = replicate(g.vertexNum, UNVISITED)
  val (_,resTree) = from(startFront, initTree)
                         .until(stopCondition)(doStep)
  return resTree
}
```

Let's do the same trick with MST

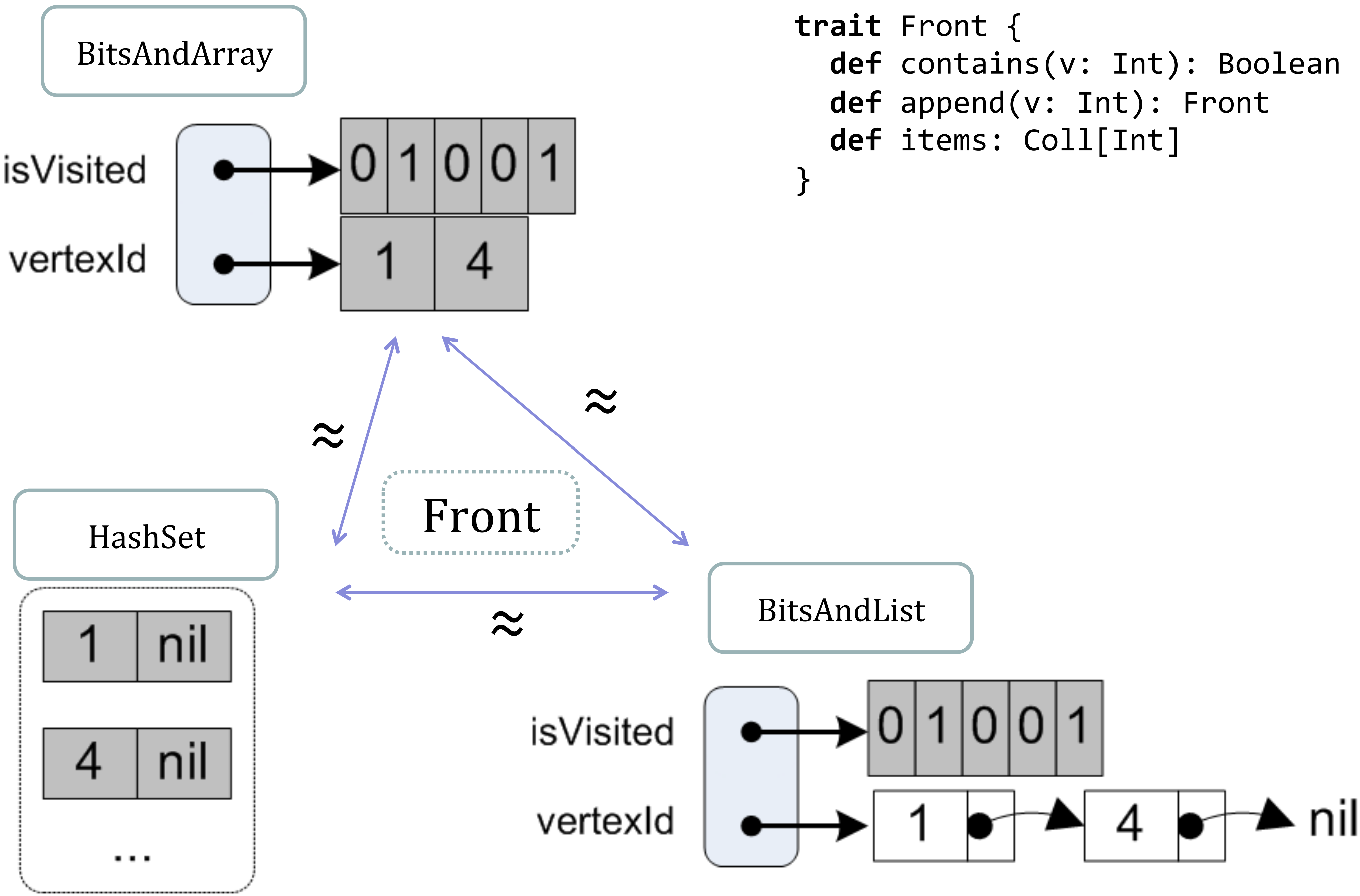
Which Graph representation is better?

```
trait Graph {
  def vertexNum: Int
  def outDegrees: Coll[Int]
  def neighbors: Coll[Int]
  def nonZeroEdgeValues: Coll[Double]
  def edgeValues: Coll[Double]
}
```



Which Front representation is better?

```
trait Front {
  def contains(v: Int): Boolean
  def append(v: Int): Front
  def items: Coll[Int]
}
```



Specialized versions of MST

MST.scala	FlatAdjacencyList	AdjacencyList	Adjacency Matrix
BitsAndArray	MST_{11}.scala	MST_{12}.scala	…
HashSet	…		
BitsAndList			

MST.cpp	FlatAdjacencyList	AdjacencyList	Adjacency Matrix
BitsAndArray	MST_{11}.cpp	MST_{12}.cpp	…
HashSet	…		
BitsAndList			

Specialized versions of MST

MST.scala	FlatAdjacencyList	AdjacencyList	Adjacency Matrix
BitsAndArray	✔		✔
HashSet	✔		✔
BitsAndList	✔		✔

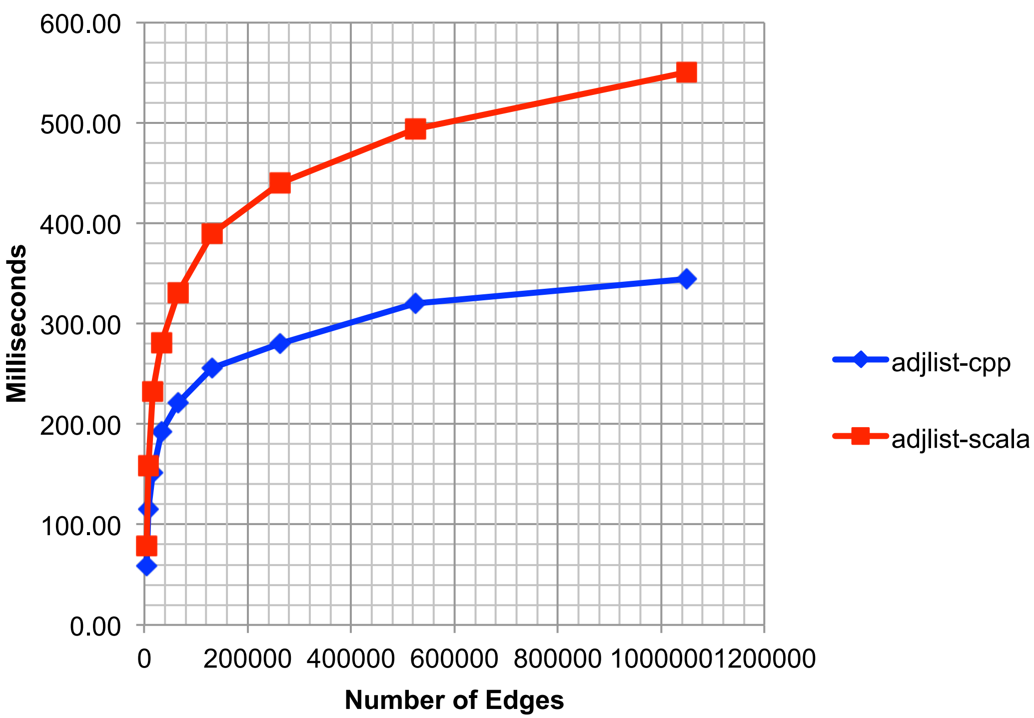
MST.cpp	FlatAdjacencyList	AdjacencyList	Adjacency Matrix
BitsAndArray	✔		✔
HashSet			
BitsAndList			

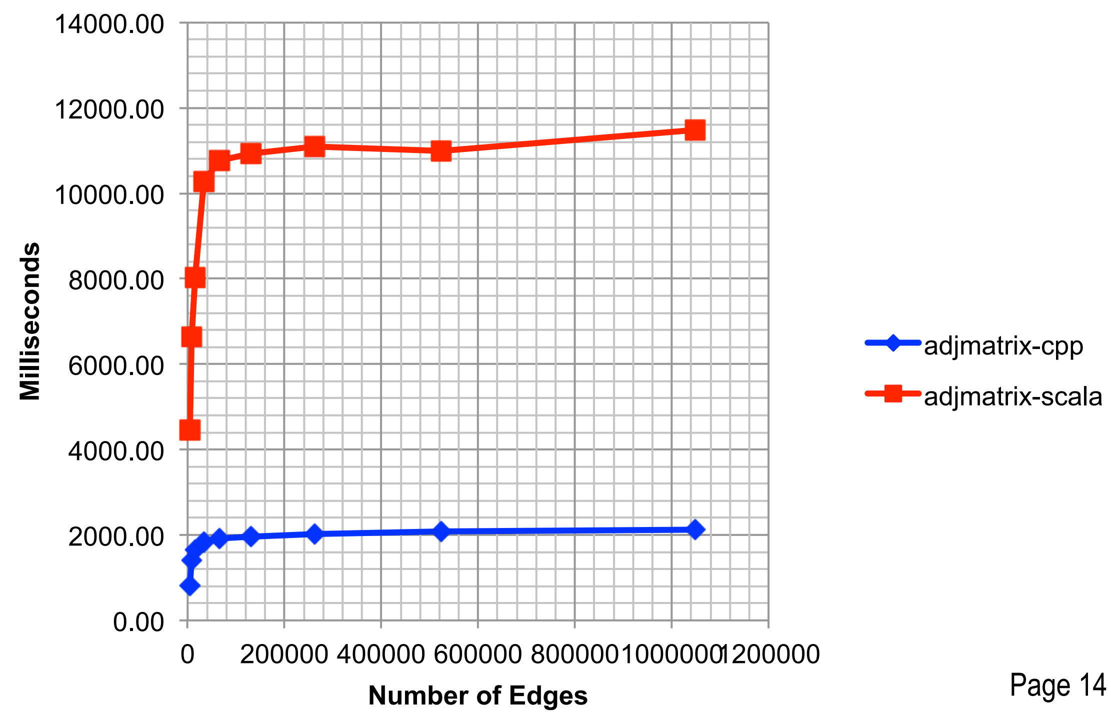
Evaluation (Scala,JVM, RMAT, scale 10, JMH)

MST.scala	FlatAdjacencyList Graph			AdjacencyMatrix Graph		
density, %	**BitsAndArray**	**BitsAndList**	**HashMap**	**BitsAndArray**	**BitsAndList**	**HashMap**
0,39	79,01	73,58	154,56	4455,39	4752,65	25911,33
0,78	158,85	161,25	242,86	6636,77	7857,11	26880,82
1,56	232,62	227,48	316,46	8025,18	9527,89	26514,70
3,13	280,84	290,60	396,36	10275,31	9753,09	26791,11
6,25	330,87	354,82	482,68	10768,81	9936,64	25896,51
12,50	389,83	429,32	556,37	10920,80	10391,19	25913,30
25,00	439,71	480,34	624,44	11083,35	10899,72	26325,43
50,00	493,87	538,96	717,31	10992,42	11516,70	26520,92
100,00	550,48	589,43	773,38	11481,12	11623,07	25533,62

Evaluation (JNI/C++, RMAT, scale 10, JMH)

MST.scala	FlatAdjacencyList Graph, BitsAndArray Front		AdjacencyMatrix Graph, BitsAndArray Front	
density, %	JNI/C++	JNI/Scala,JVM	C++	Scala, JVM
0,39	58,93	79,01	825,33	4455,39
0,78	115,15	158,85	1418,79	6636,77
1,56	151,77	232,62	1663,74	8025,18
3,13	192,11	280,84	1835,39	10275,31
6,25	221,15	330,87	1918,54	10768,81
12,50	255,74	389,83	1962,44	10920,80
25,00	279,78	439,71	2017,17	11083,35
50,00	319,85	493,87	2083,04	10992,42
100,00	344,02	550,48	2131,32	11481,12





Method summary

1. Develop an algorithm using domain-specific abstract data types (e.g. `Graph`, `Front`, `Collection`, etc.)

2. Identify *isomorphic* representations of domain objects (e.g. `AdjListGraph`, `AdjMatrGraph`, etc.)

3. Implement domain-specific interfaces using concrete representations (e.g. `AdjListGraph implements Graph`, `etc.`) and primitives of the Core language

4. Automatically specialize the algorithm with respect to all the alternative data representations

5. Compile specializations into target platform (Java, C++) using domain-specific compilation and select the best one.

Further reading

1. A. Slesarenko, A. Filippov, and A. Romanov, "First-class isomorphic specialization by staged evaluation," in Proceedings of the 10th ACM SIGPLAN workshop on Generic programming - WGP '14, 2014, pp. 35–46.

2. A.Slesarenko, Lightweight Polytypic Staging of DSLs in Scala, META'12

3. A.Slesarenko. Lightweight Polytypic Staging: a new approach to an implementation of Nested Data Parallelism in Scala. The Third Annual Scala Workshop, April 17–18, 2012, London, UK.

4. A.Slesarenko. Scalan: polytypic library for nested parallelism in Scala. Preprint 22. Keldysh Institute of Applied Mathematics, Moscow. 2011.

5. LMS (http://scala-lms.github.io/)

Download these and related Scalan publications on my home page (http://pat.keldysh.ru/~slesarenko/)

Join us or get involved

1. Check out source code at **github.com/scalan**

2. Ask questions on Google Group
   https://groups.google.com/d/forum/scalan

3. Follow us on twitter (**@avslesarenko, @alexey_r**)

Thank you

github.com/scalan