Haydi Documentation

Release 1.0

Haydi team

Dec 04, 2017

User guide

CHAPTER 1

Overview

Haydi (Haystack diver) is a **framework for generating discrete structures**. It provides a way to define a structure from basic building blocks (e.g. Cartesian product, mappings) and then enumerate all elements, all non-isomorphic elements, or generate random elements.

- Pure Python implementation (Python 2.7+, PyPy supported)
- MIT license
- Sequential or distributed computation (via dask/distributed)

1.1 Example of usage

- Let us define **directed graphs on two vertices** (represented as a set of edges):

```
>>> import haydi as hd
>>> nodes = hd.USet(2, "n")  # A two-element set with (unlabeled) elements {n0, n1}
>>> graphs = hd.Subsets(nodes * nodes)  # Subsets of a cartesian product
```

- Now we can **iterate all elements**:

```
>>> list(graphs.iterate())
[{}, {(n0, n0)}, {(n0, n0), (n0, n1)}, {(n0, n0), (n0, n1), (n1, n0)}, {(n0,
# ... 3 lines removed ...
n1)}, {(n1, n0)}, {(n1, n0), (n1, n1)}, {(n1, n1)}]
```

- or **iterate all non-isomorphic ones**:

```
>>> list(graphs.cnfs())  # cnfs = canonical forms
[{}, {(n0, n0)}, {(n0, n0), (n1, n1)}, {(n0, n0), (n0, n1)}, {(n0, n0), (n0,
n1), (n1, n1)}, {(n0, n0), (n0, n1), (n1, n0)}, {(n0, n0), (n0, n1), (n1, n0),
(n1, n1)}, {(n0, n0), (n1, n0)}, {(n0, n1)}, {(n0, n1), (n1, n0)}]
```

- or **generate random instances**:

```
>>> list(graphs.generate(3))
[{(n1, n0)}, {(n1, n1), (n0, n0)}, {(n0, n1), (n1, n0)}]
```

- Haydi supports standard operations like **map, filter and reduce**:

```
>>> op = graphs.map(lambda g: len(g)).reduce(lambda x, y: x + y)
# Just a demonstration pipeline; nothing usefull
>>> op.run()
```

- We can run it transparently as a **distributed application**:

```
>>> from haydi import DistributedContext
# We are now assuming that dask/distributed is running at hostname:1234
>>> context = DistributedContext("hostname", 1234)
>>> op.run(ctx=context)
```

CHAPTER 2

Example: Černy's conjecture

The following example shows how to use Haydi for verifying Černy's conjecture on bounded instances. The conjecture states that the length of a minimal reset word is bounded by $(n-1)^2$ where n is the number of states of the automaton. The reset word is a word that send all states of the automaton to a unique state.

The example program find the maximal length of a minimal reset word for automata of a given size. The full source code is in *examples/cerny/cerny.py* in the repository.

The following approach is a simple one, just a few lines of code, without any sophisticated optimization. It takes around two minutes (in PyPy) in the sequential mode to verify the conjecture for automata with five states. It probably needs many hours to check automata with six states. The state of the art result is verifying the conjecture for automata for more than 14 states, but it needs some clever optimizations that are out of scope of this example.

First, we describe deterministic automata by their transition functions (mapping from pair of state and symbol to a new state):

```
>>> import haydi as hd
>>> n_states = 4   # Number of states
>>> n_symbols = 2  # Number of symbols in alphabet

>>> states = hd.USet(n_states, "q")  # set of states q0, q1, ..., q_{n_states-1}
>>> alphabet = hd.USet(n_symbols, "a")  # set of symbols a0, ..., a_{a_symbols-1}

# All Mappings (states * alphabet) -> states
>>> delta = hd.Mappings(states * alphabet, states)
```

Now we can create a pipeline that goes through all non-isomorphic automata and finds maximum among lengths of their minimal reset word:

```
>>> pipeline = delta.cnfs().map(check_automaton).max(size=1)
>>> result = pipeline.run()

>>> print ("The maximal length of a minimal reset word for an "
...        "automaton with {} states and {} symbols is {}.".
...        format(n_states, n_symbols, result[0]))
```

The function `check_automaton` takes an automaton (as a transition function) and returns the length the minimal reset word or 0 when there is no such word. It is just a simple breath-first search on the set of states:

```
from haydi.algorithms import search

# Let us precompute some values that will be repeatedly used
init_state = frozenset(states)
max_steps = (n_states**3 - n_states) / 6
# Known result is that we do not need more than (n^3 - n) / 6 steps

def check_automaton(delta):
      # This function takes automaton as a transition function and
      # returns the minimal length of synchronizing word or 0 if there
      # is no such word

      def step(state, depth):
          # A step in bread-first search; gives a set of states
          # and return a set reachable by one step
          for a in alphabet:
              yield frozenset(delta[(s, a)] for s in state)

      delta = delta.to_dict()
      return search.bfs(
          init_state,  # Initial state
          step,        # Function that takes a node and returns the followers
          lambda state, depth: depth if len(state) == 1 else None,
                       # Run until we reach a single state
          max_depth=max_steps,  # Limit depth of search
          not_found_value=0)    # Return 0 when we exceed depth limit
```

CHAPTER 3

Installation

Haydi is a pure Python package for Python 2.7+.

3.1 Basic installation

Haydi itself can be installed simply with `pip`:

```
pip install git+https://github.com/spirali/haydi.git
```

3.2 Distributed run

For *distributed run* you need to install dask/distributed:

```
pip install distributed
```

3.3 PyPy

Haydi is fully compatible with PyPy that can provide a substantial speed-up.

CHAPTER 4

Domains

This section introduces the core structure of *Haydi*: *domains* and basic operations with them. Advanced domains and canonical forms are covered in a separate section: *Canonical forms*.

4.1 Elementary Domains

One of basic structures in Haydi is `Domain` that represents a generic collection of arbitrary objects. The main operation with domains is to provide a method for iteration and random generation of elements in domain. Domains are composable, i.e. more complex domains can be created from the simpler ones.

There are six *elementary domains* shipped with Haydi: `Range` (range of integers), `Values` (domain of explicitly listed Python objects), `Boolean` (two-element domain), and `NoneDomain` (a domain containing only one element: `None`), `USet`, and `CnfValues`.

Domains `USet` and `CnfValues` are little bit special and they are designed for enumerating non-isomorphic structures. The topic is covered in *Canonical forms*; these domains are not used in this section.

Examples:

```
>>> import haydi as hd

>>> hd.Range(4)  # Domain of four integers
<Range size=4 {0, 1, 2, 3}>

>>> hd.Values(["Haystack", "diver"])
<Values size=2 {'Haystack', 'diver'}>

>>> hd.Boolean()
<Boolean size=2 {False, True}>

>>> hd.NoneDomain()
<NoneDomain size=1 {None}>
```

4.2 Composition

New domains can be created by composing existing ones. There are the following compositions: *product*, *sequences*, *subsets*, *mappings*, and *join*.

4.2.1 Cartesian product $(A \times B)$

`Product` creates a domain of all ordered tuples; for example:

```
>>> import haydi as hd
>>> a = hd.Range(2)
>>> b = hd.Values(("a", "b", "c"))
>>> hd.Product((a, b))
<Product size=6 {(0, 'a'), (0, 'b'), (0, 'c'), (1, 'a'), ...}>
```

alternatively, the same thing can be written by using the infix operator `*`:

```
>>> a * b
<Product size=6 {(0, 'a'), (0, 'b'), (0, 'c'), (1, 'a'), ...}>
```

The product can be created on more than two domains:

```
>>> hd.Product((a, b, hd.Values["x", "y"]))
<Product size=12{(0, 'a', 'x'), (0, 'a', 'y'), (0, 'b', 'x'), ...}>

>>> a * b * hd.Values(["x", "y"])
<Product size=12{(0, 'a', 'x'), (0, 'a', 'y'), (0, 'b', 'x'), ...}>
```

Note: Generally, `a * b` equals to `hd.Product((a, b))`. However, there is one exception when `a` is also product. The expression `hd.Product((x, y)) * b` is equal to `hd.Product((x, y, b))` (not `hd.Product(hd.Product(x, y), b)`). The reason is to enable defininining n-ary tuples by multiplication. If you want to avoid this behavior and define "product in product", then explicitly use `hd.Product` instead of `*`.

4.2.2 Sequences (A^n)

`Sequences` is a shortcut for a product over the same domain. Sequences of a given length can be defined as:

```
>>> import haydi as hd
>>> a = hd.Range(2)
>>> hd.Sequences(a, 3)  # Sequences of length 3
<Sequences size=8 {(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), ...}>
```

or sequences with a length in a given range:

```
>>> hd.Sequences(a, 0, 2)  # Sequences of length 0 to 2
<Sequences size=7 {(), (0,), (1,), (0, 0), ...}>
```

Sequences of a fixed length can also be created by the ** operator on a domain:

```
>>> hd.Range(2) ** 3
<Sequences size=8 {(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), ...>
```

4.2.3 Subsets ($\mathcal{P}(A)$)

The `Subsets` contains subsets from elements of a given domain; the following example creates the power set:

```
>>> import haydi as hd
>>> hd.Subsets(hd.Range(2))
<Subsets size=4 {{}, {0}, {0, 1}, {1}}>
```

When a single argument is provided, it is used to limit subsets to a given size:

```
>>> hd.Subsets(hd.Range(3), 2)  # Subsets of size 2
<Subsets size=3 {{0, 1}, {0, 2}, {1, 2}}>
```

Two arguments limit the subsets to a size in a given range:

```
>>> hd.Subsets(hd.Range(3), 0, 1)  # Subsets of size between 0 and 1
<Subsets size=4 {{}, {0}, {1}, {2}}>
```

Note: Type of elements created by `Subsets` is *not* the standard Python `set`, but `haydi.Set`. For more information, see *Set and Map*. This behavior can be overridden by argument `set_class`:

```
>>> hd.Subsets(hd.Range(2), set_class=frozenset)
<Subsets size=4 {frozenset([]), frozenset([0]), frozenset([0, 1]), ...}>
```

4.2.4 Mappings ($A \to B$)

The domain `Mappings` contains all mappings from a domain to another domain:

```
>>> import haydi as hd
>>> a = hd.Range(2)
>>> b = hd.Values(["a", "b"])
>>> hd.Mappings(a, b)
<Mappings size=4 {{0: 'a'; 1: 'a'}, {0: 'a'; 1: 'b'}, {0:  ... a'}, ...}>
```

Note: Type of elements created by `Mappings` is *not* the standard Python `dict`, but `haydi.Map`. For more information, see *Set and Map*. This behavior can be overridden by argument `map_class`:

```
>>> hd.Mappings(a, b, map_class=dict)
<Mappings size=4 {{0: 'a', 1: 'a'}, {0: 'a', 1: 'b'}, {0:  ... a'}, ...}>
```

4.2.5 Join ($A \uplus B$)

`Join` operation creates a new domain that contains elements of all given domains (disjoint union):

```
>>> import haydi as hd
>>> a = hd.Range(2)
>>> b = hd.Values(["abc", "ikl", "xyz"])
>>> c = hd.Values([123])

>>> hd.Join((a, b, c))
<Join size=6 {0, 1, 'abc', 'ikl', ...}>
```

The same behavior can be also achieved by + operator on domains:

```
>>> a + b + c
<Join size=6 {0, 1, 'abc', 'ikl', ...}>
```

Note that `Join` does not collapse the same elements in the joined domains:

```
>>> a = hd.Range(2)
>>> b = hd.Range(3)
>>> a + b
<Join size=5 {0, 1, 0, 1, 2}>
```

Let us make now a small detour: Each domain can create a random element by calling `generate_one()`:

```
>>> a = hd.Range(2)
>>> b = hd.Values(["abc", "ikl", "xyz"])
>>> c = hd.Values([123])
>>> d = a + b + c
>>> d.generate_one()
"ikl"
```

By default, domains return each element with the same probability and `Join` is not an exception. Therefore, each element of `d` has probablity 1/6 to be returned by `generate_one()` (`d` has six elements).

This can be changed by `ratios` argument:

```
>>> d2 = hd.Join((a, b, c), ratios=(1, 1, 1))
```

First, we choose with the same probability (1:1:1) from which subdomain we want to pick an element and `generate_one()` is called on the selected domain. Therefore 123 will occur with probability 1/3; "ikl" has probability 1/9.

4.3 Laziness of domains

A domain is generally a lazy object that does not eagerly construct its elements. Therefore if we use code like this:

```
>>> import haydi as hd
>>> a = hd.Range(1000000)
>>> a * a * a
<Product size=1000000000000000000 {(0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 0, 3), ...}>
```

we obtain the result instantly, it only instantiates first few objects for the repr string. The ways how to instantiate elements from a domain is explained in *Pipeline*.

4.4 Transformations

Another way of creating new domains is applying a transformation on an existing domain. There are two basic transformations: **map** and **filter**.

4.4.1 Map

Map transformation takes elements of a domain and applies a function to each element to create a new domain:

```
>>> a = hd.Range(4).map(lambda x: x * 10)
>>> a
<MapTransformation size=4 {0, 10, 20, 30}>
```

The resulting object is again a domain. For example we can make a product of it:

```
>>> a * a
<Product size=16 {(0, 0), (0, 10), (0, 20), (0, 30), ...}>
```

4.4.2 Filter

Filter transformation creates a new domain by removing some elements from an existing domain. What elements are removed is configured by providing a function that is called for each element and should return True/False. When the function returns True, then the element is put into the new domain.

```
>>> hd.Range(10).filter(lambda x: x % 2 == 0 and x > 5)
<FilterTransformation size=10 filtered>
```

As we can see, the returned repr string is a different from what we have seen so far. The flag 'filtered' means that domain contains a filter transformation and the size argument is not exact but only an upper bound. The reason is that to obtain a real size we need to apply the filter function to each element (which would require going through a potentially very large domain).

If we transform the domain into the list, we force the evaluation of the domain and obtain:

```
>>> list(a)
[6, 8]
```

The 'filtered' flag is propaged during the composition of domains. When a domain is created by composing at least one filtered domain, it is also filtered:

```
>>> p = a * a
<Product size=100 filtered>
```

4.5 Names

It is possible to provide a name for a domain as an argument in the domain constructor. This name serves only for debugging purposes. For example:

```
>>> import haydi as hd
>>> a = hd.Range(10, name="MyRange")
>>> a
<MyRange size=10 {0, 1, 2, 3, ...}>
>>> a.name
'MyRange'

>>> a = hd.Range(10)
>>> hd.Product((a, a), name="MyProduct")
<MyProduct size=100 {(0, 0), (0, 1), (0, 2), (0, 3), ...}>
```

CHAPTER 5

Pipeline

This section contains a description of working with elements of domains. The main message of this section is that there are three basic methods of creating a stream of elements from a domain:

- `iterate()` – stream of all elements in domain
- `cnfs()` – stream of all canonical elements in domain
- `generate()` – stream of random elements from domain

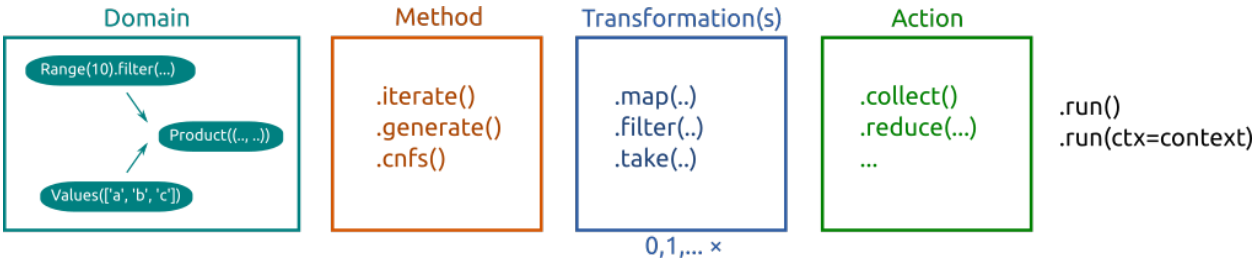
The rest of this section describes the whole machinery in more detail. If you are interested in sequential computations only, and you want to handle the stream manually, you can just directly use Python iterators as follows:

```
>>> import haydi as hd
>>> for x in hd.Subsets(hd.Range(10)).generate(5):  # print five random subsets
>>>     print x
```

The purpose of the pipeline is to simplify some common operations and enable transparent distributed computations.

5.1 Overview

The whole pipeline is composed of the following elements:



- *Domain* – a domain as described in the previous section
- *Method* – how to take elements from the domain into the stream

- *Transformations* – transformations of the stream
- *Action* – final aggregation of results
- *Run* – the actual invocation of the pipeline

5.2 Method

There are three methods how we can walk through a domain: iterate, iterate through canonical forms and random generation.

5.2.1 Iterate

A pipeline that iterates through all elements is created by method `iterate()`:

```
>>> import haydi as hd
>>> domain = hd.Range(2) * hd.Range(2)
>>> domain.iterate()
<Pipeline for Product: method=iterate action=Collect>
```

Calling `iterate()` on a domain creates a pipeline object. Moreover, we can also see that the default action is *Collect*. This action simply takes all elements and put them into the list. More details about actions can be found in Section *Actions*.

The pipeline is a lazy object and no elements are actually constructed. To run the pipeline, we need to call `run()` method:

```
>>> domain.iterate().run()
[(0, 0), (0, 1), (1, 0), (1, 1)]
```

The `iterate()` method iterates through all elements in the domain. It is guaranteed that each element in the domain occurs in the stream in the same number of occurrences as in the domain. The actual order of elements in the stream is *not* guaranteed.

5.2.2 Canonical forms

Iterating over canonical elements is a more complex operation, hence there is a dedicated section about this topic: *Canonical forms*.

5.2.3 Random elements

A pipeline that generates random elements from a domain is created by method `generate(count=None)`, where the optional parameter `count` specifies the number of generated elements:

```
>>> domain = hd.Range(2) * hd.Range(2)
>>> domain.generate(5)
[(0, 1), (0, 0), (1, 0), (0, 0), (1, 1)]
```

By default, all elements are generated with the same probability, however it can be configured in some places. See the API documentation for `Join`.

When the argument of `generate` is `None`, then we obtain an infinite pipeline of random instances. It usually makes sense in combination with a filter and setting a limit after the filter.

For example, the following code generates 10 pairs whose sum is 11:

```
>>> domain = hd.Range(10) * hd.Range(10)
>>> domain.generate().filter(lambda x: x[0] + x[1] == 11).take(5).run()
[(3, 8), (5, 6), (9, 2), (3, 8), (6, 5)]
```

Transformation `take(5)` limits the pipeline for the first five elements. As an exercise we left what happens when we put `5` as the argument for `generate` and remove the `take`. TODO

5.3 Transformations

The current version offers three *pipeline transformations*:

- `map(fn)` – maps the function `fn` on each element that goes through the pipeline
- `filter(fn)` – filters elements in the pipeline according to the provided function
- `take(count)` – takes only first `count` elements from pipeline

At the first sight, there is an overlap between transformations on domains and in the pipeline. In fact, they have in many cases completely the same effect:

```
>>> domain = hd.Range(5)
>>> domain.map(lambda x: x * 10).iterate().run()  # Create a new domain and then␣
↪iterate
[0, 10, 20, 30, 40]
>>> domain.iterate().map(lambda x: x * 10).run()  # Transformation in pipeline
[0, 10, 20, 30, 40]
```

So why distinguish transformations in pipelines and on domains? The reason is that in the pipeline, we know that process of the domain creation is completed and have more freedom for additional features and optimizations. We already have a stream of elements; therefore, we can introduce `take` transformation. Moreover, the pipeline transformations do not have limitation in case of *Strict domains* that are important in the usage of `cnfs()`.

For performance reasons, pipeline transformations provide more opportunities for efficient distributed computations. Therefore, Haydi prefers *map* and *filter* transformations as pipeline transformations rather than domain transformations. For this reason, Haydi automatically moves last transformations on domains to the pipeline; therefore, the above example actually creates the same pipeline (with one pipeline transformation):

```
>>> domain.map(lambda x: x * 10).iterate()
<Pipeline for Range: method=iterate ts=[MapTransformation] action=Collect>
>>> domain.iterate().map(lambda x: x * 10)
<Pipeline for Range: method=iterate ts=[MapTransformation] action=Collect>
```

Of course 'inner' domain transformations cannot be moved. For example the following code creates a pipeline without any transformation (the transformation remains hidden inside the domain composition):

```
>>> domain = hd.Subsets(hd.Range(3).map(lambda x: x * x))
>>> domain.iterate().run()
[{}, {0}, {0, 1}, {0, 1, 4}, {0, 4}, {1}, {1, 4}, {4}]
>>> domain.iterate()
<Pipeline for Subsets: method=iterate action=Collect>
```

5.4 Actions

Action is a terminal operation on a stream of elements. The list of operations follows; more details can be found in API documentation of `Pipeline`.

5.4.1 Collect

Action *collect* creates a list from the stream:

```
>>> hd.Range(5).iterate().collect().run()
[0, 1, 2, 3, 4]
```

The *collect* is the default action; therefore, the above code is equivalent to:

```
>>> hd.Range(5).iterate().run()
[0, 1, 2, 3, 4]
```

5.4.2 First

Action *first* takes the first element from the stream. If the stream is empty it returns the provided argument (the default is `None`).

```
>>> hd.Range(5).iterate().first().run()
0
>>> hd.Range(5).filter(lambda x: x > 10).first().run()
None
>>> hd.Range(5).filter(lambda x: x > 10).first("no value").run()
'no value'
```

5.4.3 Reduce

Action *reduce* applies a binary operation on elements of the stream:

```
>>> hd.Range(10).reduce(lambda x, y: x + y).run()
45
```

You can optionally specify an initial value:

```
>>> hd.Range(10).reduce(lambda x, y: x + y, -3).run()
42
```

It is assumed by default that the operation is associative, if that is not true, you have to explicitly specify it:

```
>>> hd.Range(10).reduce(lambda x, y: x - y, 100, associative=False).run()
55
```

5.4.4 Max

Action *max* gathers maximal elements in the stream, optionally it can take a function that extracts a value from the element that is used for comparison. The second optional argument specifies the limit of maximal elements. No more than the limit number of elements is returned; the rest of maximal elements is thrown away. Which maximal elements

are thrown away and what are returned is not specified. If the value of the second argument is `None` (default) then all maximal elements are returned:

```
>>> domain = hd.Range(5) * hd.Range(5)

>>> domain.max().run()
[(4, 4)]

>>> domain.max(lambda x: x[0]).run()  # Maximum in the first element in the pair
[(4, 0), (4, 1), (4, 2), (4, 3), (4, 4)]

>>> domain.max(lambda x: x[0], 2).run()  # At most two maximal elements
[(4, 0), (4, 1)]
```

5.4.5 Groups

Action *groups* divides elements in the stream into groups according to a key. The method takes a function that is applied on each element to obtain the key.

```
>>> hd.Range(10).groups(lambda x: x % 3).run()
{0: [0, 3, 6, 9], 1: [1, 4, 7], 2: [2, 5, 8]}
```

Optionally, it takes an integer argument that limits the size of groups. No more than the limit number of elements is returned for each group. What elements in the group are thrown away and what are returned is not specified.

```
>>> hd.Range(10).groups(lambda x: x % 3, 2).run()
{0: [0, 3], 1: [1, 4], 2: [2, 5]}
```

5.4.6 Groups_counts

This is an extension of Groups action that also returns the total number of elements including the elements that was thrown away. The total number of elements is returned at the first index of the lists:

```
>>> hd.Range(10).groups_counts(lambda x: x % 3, 2).run()
{0: [4, 0, 3], 1: [3, 1, 4], 2: [3, 2, 5]}
```

5.5 Run

The `run(ctx=None, timeout=None, otf_trace=False)` method invokes the pipeline. By default, it creates and executes a sequential computation without any time limit. This can be changed by arguments.

The `ctx` parameter defines a context used for the execution of the the pipeline. Providing an instance of a distributed context, makes the pipeline parallel and distributed, see *distributed computation*.

Parameter `timeout` expects `float` (a number of seconds) or a `timedelta` object. This defines the maximum time of the computation. If the allocated time runs out, the computation is stopped and a partial result is returned.

5.6 Shortcuts

To make the code more concise, there are the following defaults defined for the pipeline:

- Method: `iterate`
- Transformations: None
- Action: `collect`
- Run: `run()`

Therefore, we can call `.run()` directly on domain and obtain the same results as using `.iterate().collect().run()`. It automatically creates the default pipeline.

In the same manner, we can also directly call actions on a domain. It creates a pipeline with `iterate()` method.

Examples:

```
>>> hd.Range(5).run()  # .iterate().collect() is used
[0, 1, 2, 3, 4]

>>> hd.Range(5).max().run()  # .iterate() is used
[4]
```

When we create an iterator from a domain or a pipeline, we obtain an iterator to the result of pipeline where missing elements are filled by defaults:

```
>>> list(hd.Range(5))
[0, 1, 2, 3, 4]

>>> list(hd.Range(5).map(lambda x: x * x))
[0, 1, 4, 9, 16]

>>> list(hd.Range(5).max())
[4]
```

5.7 Immutability of pipelines

Pipelines are immutable objects (as same domains); therefore, calling methods on them actually creates new objects. It is thus safe to reuse them:

```
>>> pipeline = hd.Range(5).iterate()
>>> pipeline.take(2).run()
[0, 1]
>>> pipeline.max().run()
[4]
```

CHAPTER 6

Set and Map

Domains distributed with Haydi use standard Python types almost in all places for example: `Range` creates a domain where elements have type `int`, `Product` and `Sequences` use `tuple` as elements in the resulting domains. The exceptions are `Subsets` that use `haydi.Set` and `Mappings` with `haydi.Map`, even though a natural choice would be a standard `set` and `dict`. The reason is a performance optimization during generation (mainly in generating *Canonical forms*).

If you need, both types can be simply converted into the standard ones:

```
>>> import haydi as hd
>>> a = hd.Range(2)

>>> hd.Subsets(a).map(lambda s: s.to_set())  # Creates standard Python sets
<MapTransformation size=4 {set([]), set([0]), set([0, 1]), set([1])}>

>>> hd.Subsets(a).map(lambda s: s.to_frozenset())  # Creates standard Python␣
↪frozensets
<MapTransformation size=4 {frozenset([]), frozenset([0]), frozenset([0, , ...}>

>>> hd.Mappings(a, a).map(lambda m: m.to_dict())  # Creates standard Python dicts
<MapTransformation size=4 {{0: 0, 1: 0}, {0: 0, 1: 1}, {0: 1, 1: 0}, {0:, ...}>
```

Note: Classes `Set` and `Map` are not designed for frequent searching. If you need it, please convert them to set/dict. From this reasons they do not intentionally implement methods `__in__` and `__getitem__` to avoid accident usage theses class instead of set/dict. For occasional lookup, there are methods contains/get in these classes.

This is still a subject of discussions if we want to introduce `__in__` and `__getitem__` method for these classes.

CHAPTER 7

Canonical forms

This section covers a feature that serves to iterate only non-isomorphic elements in a domain. It is based on iterating over canonical forms – one element for each equivalence class of isomorphism.

Basic building blocks for the whole machinery are *Atoms* and *USets*, that allows to define bijections between elements.

7.1 Atoms and USets

Domain `USet` contains a finite number of instances of class `Atom`. When a new USet is created a number of atoms and a name have to be specified. With a new USet new atoms are created. Each atom remembers its index number and parent USet that created it:

```
>>> import haydi as hd
>>> uset = hd.USet(3, "a")   # Create a new USet
>>> uset
<USet id=1 size=3 name=a>
>>> list(uset)               # Atoms in uset
[a0, a1, a2]
>>> a0, a1, a2 = list(uset)
>>> a0.index
0
>>> a0.parent                # Each Atom remembers its parent USet
<USet id=1 size=3 name=a>
```

From the perspective of pipeline methods `iterate()` and `generate()`, USet behaves as a kind of fancy `Range` that wraps numbers into special objects.

The main difference between `Range` and `USet` arises when we introduce *isomorphism*. Let us remind that two (discrete) objects are *isomorphic* if there exits a *bijection* between them that preserves the structure.

In our case, we are establishing bijection between atoms and two objects are isomorphic when we can obtain one from another by replacing atoms according to the bijection.

Let us show some examples that use `haydi.is_isomorphic()` which checks isomorphism:

```
>>> a0, a1, a2 = hd.USet(3, "a")

>>> hd.is_isomorphic(a0, a1)
True  # Because there is bijection: a0 -> a1; a1 -> a0; a2 -> a2

>>> hd.is_isomorphic((a0, a1), a1)
False  # No mapping between atoms can bring us from a tuple to an atom

>>> hd.is_isomorphic((a0, a1), (a1, a2))
True  # Because there is mapping: a0 -> a1; a1 -> a2; a2 -> a0

>>> hd.is_isomorphic((a0, a0), (a0, a1))
False  # The explanation below
```

The bijection between objects in the last case cannot exists. The first tuple represents a pair of the same object and "renaming" a0 to anything else preserves this property. The second one represents a pair of two different atoms (even from the same USet) and any renaming cannot achieve the property of the first one (any mapping containing `a0 -> a0; a1 -> a0` is not bijective)

7.2 Atoms from different USets

Two atoms from different USets cannot be renamed to each other; i.e. they are never isomorphic. It can be seen as each USet and its atoms have a different color.

```
>>> a0, a1 = hd.USet(2, "a")
>>> b0, b1 = hd.USet(2, "b")
```

```
>>> hd.is_isomorphic(a0, b0)
False  # Map containing a0 -> b0 is not allowed
```

```
>>> hd.is_isomorphic((a0, b1), (a1, b0))
True  # There is bijection: a0 -> a1; a1 -> a0; b0 -> b1; b1 -> b0
```

The bijection in the second case is correct, since each atom has an image from its parent USet.

Note: The name of an USet serves only for the debugging purpose and has no impact on behavior. Creating two USets with the same name still creates two disjoint sets of atoms with their own parents.

7.3 Basic objects

So far, we have seen atoms and tuples of atoms in the examples. However, the whole machinery around isomorphisms is implemented for objects that we call *basic objects*; they are inductively defined as follows:

- atoms, integers, strings, True, False, and None are basic objects
- a tuple of basic objects is a basic object
- `haydi.Set` of basic objects is a basic object
- `haydi.Map` where keys and values are basic objects is a basic object

Examples:

```
>>> a0, a1, a2 = hd.USet(3, "a")

>>> hd.is_isomorphic((a0, 1), (a0, 2))
False  # Renaming is defined only for atoms, not for other objects

>>> hd.is_isomorphic((a0, 1), (a1, 1))
True  # Bijection: a0 -> a1; a1 -> a0; a2 -> a2

>>> hd.is_isomorphic(hd.Set((a0, a1)), hd.Set((a2, a0)))
True  # Bijection: a0 -> a0; a1 -> a2; a2 -> a1
```

7.4 Canonical forms

Since we are interested only in finite (basic) objects, they contain only finitely many atoms, so there are only finitely many bijections (recall that USets are finite). Therefore, each class of equivalence induced by isomorphism is also finite.

In Haydi, there there is a fixed linear ordering of all basic objects defined by `haydi.compare()`. Since each isomorphic class is finite, hence each class has the smallest element according this ordering. We call this element as a *canonical form* of the class.

The pipeline method `cnfs()` iterates only through canonical elements in a domain; therefore, we obtain only one element for each equivalence class.

Let us show some examples:

```
>>> uset = hd.USet(3, "a")
>>> bset = hd.USet(3, "b")

>>> list(uset)  # All elements
[a0, a1, a2]

>>> list(uset.cnfs())  # Canonical forms
[a0]

>>> list(uset + bset)  # All elements
[a0, a1, a2, b0, b1, b2]

>>> list((uset + bset).cnfs())  # Canonical forms
[a0, b0]

>>> p = uset * uset
>>> list(p)  # All elements
[(a0, a0), (a0, a1), (a0, a2),(a1, a0), (a1, a1),
 (a1, a2), (a2, a0), (a2, a1), (a2, a2)]

>>> list(p.cnfs())  # Canonical forms
[(a0, a0), (a0, a1)]


>>> s = hd.Subsets(uset + bset, 2)
>>> list(s)  # All elements
[{a0, a1}, {a0, a2}, {a0, b0}, {a0, b1}, {a0, b2}, {a1, a2}, {a1, b0},
 {a1, b1}, {a1, b2}, {a2, b0}, {a2, b1}, {a2, b2}, {b0, b1}, {b0, b2}, {b1, b2}]
```

```
>>> list(s.cnfs())  # Canonical forms
[{a0, a1}, {a0, b0}, {b0, b1}]
```

7.5 Strict domains

The pipeline method `cnfs()` is allowed only for *strict* domains. *Strict domain* is a domain that contains only basic objects and is closed under isomorphism (if it contains an element, it contains also all isomorphic ones). We call it "strict", but it is usually not a problem to fulfill these criteria in practice.

All elementary domains except `Values` are always strict. (Domain `CnfValues` is a counter-part of `Values` for canonical forms; see *Domain CnfValues*). The strictness of a domain can be checked by reading its attribute `strict`:

```
>>> hd.Range(10).strict
True
```

All basic domain compositions preserve strictness if and only if all their inner domains are also strict, e.g.:

```
>>> domain = hd.Subsets(hd.Range(5) * hd.USet(2))
>>> domain.strict
True
```

The only places where we have to be more careful are transformations and when we create a strict domain from explicit elements. These topics are covered in next two subsections.

7.6 Transformations on strict domains

Generally transformations may break strict-domain invariant. A filter may remove some elements and left some isomorphic ones. A map may even returns some non-basic objects. Therefore, a domain created by transformation is non-strict by default.

In most cases, when we want to use `cnfs()` while applying a transformation, we can simply move the transformation into the pipeline, where are no such restrictions, since in the pipeline we do not create a new domain:

```
>>> domain = hd.USet(3, "a") * hd.Range(4)

>>> list(domain.cnfs())  # This is Ok
>>> new_domain = domain.map(lambda x: SomeMyClass(x))

>>> new_domain.strict
False

>>> new_domain.cnfs()  # Throws an error

>>> domain.cnfs().map(lambda x: SomeMyClass(x))  # This is ok, map is in pipeline
```

If we really need to create a new strict domain by applying a transformation, it is now possible only with filter by the following way:

```
>>> domain = hd.USet(3, "a") * hd.Range(4)
>>> new_domain = domain.filter(lambda x: x[1] != 2, strict=True)
>>> new_domain.strict
True
```

```
>>> list(new_domain.cnfs())
[(a0, 0), (a0, 1), (a0, 3)]
```

When the filter parameter `strict` is set to `True` and the original domain is strict, then the resulting domain is still strict.

Warning: It is the user's responsibility to assure that strict filter removes all isomorphic elements. Fortunately, in practice it is usually the desired behavior of filters. However, if the rule of strict filter is broken, the behavior of `cnfs()` is undefined on such a domain.

7.7 Domain `CnfValues`

Domain `Values` creates a non-strict domain, since we cannot assure that all invariants are valid. If you want to create a strict domain from explicit elements, you can use `CnfValues`. The difference is that `CnfValues` is constructed from canonical elements and it automatically adds necessary objects into the domain to make it strict (i.e. it adds all elements isomorphic to the given canonical elements):

```
>>> uset = hd.USet(3, "a")
>>> a0, a1, a2 = uset
>>> domain = hd.CnfValues((a0, (a0, a1), "x"))

>>> list(domain.cnfs())
[a0, (a0, a1), 'x']

>>> list(domain.iterate())
[a0, a1, a2, (a0, a1), (a0, a2), (a1, a0), (a1, a2), (a2, a0), (a2, a1), 'x']
```

7.8 Public functions

This is list of public methods that may be useful when you are working with canonical forms:

- `is_canonical()` – returns `True` if and only if a given is in canonical form.
- `haydi.expand()` – returns a list of isomorphic objects to a given objects.
- `haydi.compare()` – defines a linear ordering between two objects. The exact ordering is left is unspecified, but it is guaranteed that for basic objects it stays fixed even between separated executions.
- `haydi.sort()` – sorts object according *hayd.compare*.

CHAPTER 8

Distributed computation

The pipeline computation in Haydi can be transparently executed through dask/distributed. This enables distributed computation or local parallel computation.

Switching pipeline from default serial context to distributed one, is done through providing a class to `run` method in the pipeline.

8.1 Local computation

The following example shows how to start four local dask/distributed workers and run Haydi computation on them:

```
>>> from haydi import DistributedContext
>>> dctx = DistributedContext(spawn_workers=4)
>>> hd.Range(100).map(lambda x: x + 1).collect().run(ctx=dctx)
[1, 2, 3, 4, ...]
```

The argument `spawn_workers` forces `DistributedContext` to spawn dask/distributed workers for you. Switching pipeline from default serial context to distributed one, is done through providing a distributed context to `run` method in the pipeline.

8.2 Distributed computation

If you want to distribute the computation amongst multiple computers, you first have to create a *distributed* cluster. An example of cluster setup can be found here. Once the cluster is created, simply pass the IP adress and port of the clusters' scheduler to the context:

```
>>> dctx = DistributedContext(ip='192.168.1.1', port=8787)
# connects to cluster at 192.168.1.1:8787
>>> hd.Range(100).map(lambda x: x + 1).collect().run(ctx=dctx)
[1, 2, 3, 4, ...]
```

8.3 Limitations

The nested distributed computations are not allowed, i.e. you cannot run distributed pipeline in another distributed pipeline. The following example shows the invalid case:

```
>>> def worker(x):
...     r = hd.Range(x)
...     # invalid! sequential run() must be used
...     return r.collect().run(DistributedContext(spawn_workers=4))
# THIS IS INVALID - nested distributed computations are not allowed
>>> hd.Range(100).map(worker).run(DistributedContext(spawn_workers=4))
```

CHAPTER 9

Cookbook

Cookbook of Haydi snippets for commonly occurring patterns:

9.1 Graphs

Directed graphs (subsets of pair of nodes):

```
>>> nodes = hd.USet(3, "n")
>>> graphs = hd.Subsets(nodes * nodes)
```

Undirected graphs (subsets of two-element sets):

```
>>> nodes = hd.USet(3, "n")
>>> graphs = hd.Subsets(Subsets(nodes, 2))
```

Directed graphs with labeled arcs by “x” or “y”:

```
>>> nodes = hd.USet(3, "n")
>>> graphs = hd.Subsets(nodes * nodes * hd.Values(("x", "y"))
```

Undirected graphs with 2 “red” and 3 “blue” vertices:

```
>>> nodes = hd.USet(2, "red") + hd.USet(3, "blue")
>>> graphs = hd.Subsets(Subsets(nodes, 2))
```

9.2 Automata

Deterministic finite automaton with more accepting states:

```
>>> SIMPLE TODO
```

Deterministic one-counter automata:

```
>>> SIMPLE TODO
```

Deterministic push-down automata:

```
>>> SIMPLE TODO
```

CHAPTER 10

Performance tips

This section collects tips for better performace.

10.1 Materialization of domains

TODO

10.2 Step jumps

TODO

CHAPTER 11

Contact

11.1 Authors

- Stanislav Böhm <stanislav.bohm at vsb.cz>
- Jakub Beránek <berykubik at gmail.com>
- Martin Šurkovský <martin.surkovsky at gmail.com>

11.2 Acknowledgment

The development of Haydi was supported by the Grant Agency of the Czech Republic, project GAČR 15-13784S. Core hours for optimization on HPC infrastructure was provided by project OPEN-8-26 at the National Supercomputing Center IT4Innovations.