MatchPy: A Pattern Matching Library

Manuel Krebber[‡*], Henrik Bartels[‡], Paolo Bientinesi[‡]

◆

Abstract—Pattern matching is a powerful tool for symbolic computations, based on the well-defined theory of term rewriting systems. Application domains include algebraic expressions, abstract syntax trees, and XML and JSON data. Unfortunately, no lightweight implementation of pattern matching as general and flexible as Mathematica exists for Python [Pö16], [Hao14], [Sch14], [Jen15]. Therefore, we created the open source module MatchPy which offers similar pattern matching functionality in Python using a novel algorithm which finds matches for large pattern sets more efficiently by exploiting similarities between patterns.

Index Terms—pattern matching, symbolic computation, discrimination nets, term rewriting systems

Introduction

Pattern matching is a powerful tool which is part of many functional programming languages as well as computer algebra systems such as Mathematica. It is useful for many applications including symbolic computation, term simplification, term rewriting systems, automated theorem proving, and model checking. In this paper, we present a Python-based pattern matching library and its underlying algorithms.

The goal of pattern matching is to find a match substitution given a subject term and a pattern which is a term with placeholders [BN98]. The substitution maps placeholders in the pattern to replacement terms. A match is a substitution that can be applied to the pattern yielding the original subject. As an example consider the subject $f(a)$ and the pattern $f(x)$ where x is a placeholder. Then the substitution $\sigma = \{x \mapsto a\}$ is a match because $\sigma(f(x)) = f(a)$. This form of pattern matching without any special properties of function symbols is called syntactic matching. For syntactic patterns, the match is unique if it exists.

Among the existing systems, Mathematica [WR16] arguably offers the most expressive pattern matching. Its pattern matching offers similar expressiveness as Python's regular expressions, but for symbolic tree structures instead of strings. While pattern matching can handle nested expressions up to arbitrary depth, regular expressions cannot properly handle such nesting. Patterns are used widely in Mathematica, e.g. in function definitions or for manipulating terms. It is possible to define custom function symbols which can be associative and/or commutative. Mathematica also offers sequence variables which can match a sequence of terms instead of a single one. These are especially useful when

working with variadic function symbols. Mathics [Pö16] is an open source computer algebra system written in Python that aims to replicate the syntax and functionality of Mathematica.

To our knowledge, no existing work covers pattern matching with function symbols which are either commutative or associative but not both at the same time. However, there are functions with those properties, e.g. matrix multiplication or arithmetic mean. Most of the existing pattern matching libraries for Python only support syntactic patterns. Associativity, commutativity and sequence variables make multiple distinct matches possible for a single pattern. In addition, pattern matching with either associativity or commutativity is NP-complete in both cases [BKN87]. While the pattern matching in SymPy [MSP+17] can work with associative/commutative functions, it is limited to finding a single match. Nonetheless, for some applications it is interesting to find all possible matches for a pattern, e.g. because matches need to be processed further recursively to solve an optimization problem. Furthermore, SymPy does not support sequence variables and is limited to a predefined set of mathematical operations.

In many applications, a fixed set of patterns is matched repeatedly against different subjects. The simultaneous matching of multiple patterns is called many-to-one matching, as opposed to one-to-one matching which denotes matching with a single pattern. Many-to-one matching can gain a significant speed increase compared to one-to-one matching by exploiting similarities between patterns. This has already been the subject of research for both syntactic [Chr93], [Grä91], [NWE97] and associative-commutative pattern matching [KL91], [BCR93], [LM94], [BCR+95], [Eke95], [KM01], but not with the full feature set described above. Discrimination nets [BW84] are the state-of-the-art solution for many-to-one matching. Our goal is to generalize this approach to support all the aforementioned features.

In this paper, we present the open-source library for Python MatchPy which provides pattern matching with sequence variables and associative/commutative function symbols. In addition to standard one-to-one matching, MatchPy also includes an efficient many-to-one matching algorithm that uses generalized discrimination nets. First, we give an overview of what MatchPy can be used for. Secondly, we explain some of the challenges arising from the non-syntactic pattern matching features and how we solve them. Then we give an overview of how many-to-one matching is realized and optimized in MatchPy. Next, we present our experiments where we observed significant speedups of the many-to-one matching over one-to-one matching. Finally, we draw some conclusions from the experiments and propose future work on MatchPy.

---

* *Corresponding author: manuel.krebber@rwth-aachen.de*
‡ *RWTH Aachen University, AICES, HPAC Group*

Usage Overview

MatchPy can be installed using `pip` and all necessary classes can be imported from the toplevel module `matchpy`. Expressions in MatchPy consist of constant symbols and operations. For patterns, wildcards can also be used as placeholders. We use Mathematica's notation for wildcards, i.e. we append underscores to wildcard names to distinguish them from symbols.

MatchPy can be used with native Python types such as `list` and `int`. The following is an example of how the subject `[0, 1]` can be matched against the pattern `[x_, 1]`. The expected match here is the replacement `0` for `x_`. We use *next* because we only want to use the first (and in this case only) match of the pattern:

```
>>> x_ = Wildcard.dot('x')
>>> next(match([0, 1], Pattern([x_, 1])))
{'x': 0}
```

In addition to regular (dot) variables, MatchPy also supports sequence wildcards. They can match a sequence of arguments and we denote them with two or three trailing underscores for plus and star wildcards, respectively. Star wildcards can match an empty sequence, while plus wildcards require at least one argument to match. The terminology is borrowed from regular expressions where *, + and . are used for similar concepts.

```
>>> y___ = Wildcard.star('y')
>>> next(match([1, 2, 3], Pattern([x_, y___])))
{'x': 1, 'y': (2, 3)}
```

In the following, we omit the definition of new variables as they can be done in the same way. In addition to native types, one can also define custom operations by creating a subclass of the `Operation` class:

```
class MyOp(Operation):
  name = 'MyOp'
  arity = Arity.variadic
  associative = True
  commutative = True
```

The name is a required attribute, while the others are optional and influence the behavior of the operations. By default, operations are variadic and neither commutative nor associative. Nested associative operations have to be variadic and are automatically flattened. Furthermore, regular variables behave similar to sequence variables as arguments of associative functions, because the associativity allows arbitrary parenthesization of arguments:

```
>>> next(match(MyOp(0, 1, 2), Pattern(MyOp(x_, 2))))
{'x': MyOp(0, 1)}
```

The argument of commutative operations are automatically sorted. Note that patterns with commutative operations can have multiple matches, because their arguments can be reordered arbitrarily.

```
>>> list(match(MyOp(1, 2), Pattern(MyOp(x_, z_))))
[{'x': 2, 'z': 1}, {'x': 1, 'z': 2}]
```

We can use the `CustomConstraint` class to create a constraint that checks whether `a` is smaller than `b`:

```
a_lt_b = CustomConstraint(lambda a, b: a < b)
```

The lambda function gets called with the variable substitutions based on their name. The order of arguments is not important and it is possible to only use a subset of the variables in the pattern. With this constraint we can define a replacement rule that basically describes bubble sort:

```
>>> pattern = Pattern([h___, b_, a_, t___], a_lt_b)
>>> rule = ReplacementRule(pattern,
            lambda a, b, h, t: [*h, a, b, *t])
```

Operation	Symbol	Arity	Properties
Multiplication	$\times$	variadic	associative
Addition	$+$	variadic	associative, commutative
Transposition	T	unary	
Inversion	-1	unary	
Inversion and Transposition	$-T$	unary	

TABLE 1: *Linear Algebra Operations*

The replacement function gets called with all matched variables as keyword arguments and needs to return the replacement. This replacement rule can be used to sort a list when applied repeatedly with `replace_all`:

```
>>> replace_all([1, 4, 3, 2], [rule])
[1, 2, 3, 4]
```

Sequence variables can also be used to match subsequences that match a constraint. For example, we can use the this feature to find all subsequences of integers that sum up to 5. In the following example, we use anonymous wildcards which have no name and are hence not part of the match substitution:

```
>>> x_sums_to_5 = CustomConstraint(
...                          lambda x: sum(x) == 5)
>>> pattern = Pattern([___, x__, ___], x_sums_to_5)
>>> list(match([1, 2, 3, 1, 1, 2], pattern))
[{'x': (2, 3)}, {'x': (3, 1, 1)}]
```

More examples can be found in MatchPy's documentation [Kre17a].

Application Example: Finding matches for a BLAS kernel

BLAS is a collection of optimized routines that can compute specific linear algebra operations efficiently [LHKK79], [DCHH88], [DCHD90]. As an example, assume we want to match all subexpressions of a linear algebra expression which can be computed by the ?TRMM BLAS routine. These have the form $\alpha \times op(A) \times B$ or $\alpha \times B \times op(A)$ where $op(A)$ is either the identity function or transposition, and A is a triangular matrix. For this example, we leave out all variants where $\alpha \neq 1$.

In order to model the linear algebra expressions, we use the operations shown in Table 1. In addition, we have special symbol subclasses for scalars, vectors and matrices. Matrices also have a set of properties, e.g. they can be triangular, symmetric, square, etc. For those patterns we also use a special kind of dot variable which is restricted to only match a specific kind of symbol. Finally, we construct the patterns using sequence variables to capture the remaining operands of the multiplication:

```
A_ = Wildcard.symbol('A', Matrix)
B_ = Wildcard.symbol('B', Matrix)
A_is_triangular = CustomConstraint(
  lambda A: 'triangular' in A.properties)

trmm_patterns = [
  Pattern(Times(h___, A_, B_, t___),
    A_is_triangular),
  Pattern(Times(h___, Transpose(A_), B_, t___),
    A_is_triangular),
  Pattern(Times(h___, B_, A_, t___),
    A_is_triangular),
  Pattern(Times(h___, B_, Transpose(A_), t___),
    A_is_triangular),
]
```

With these patterns, we can find all matches for the ?TRMM routine within a product. In this example, M1, M2 and M3 are matrices, but only M3 is triangular:

```
>>> expr = Times(Transpose(M3), M1, M3, M2)
>>> for i, pattern in enumerate(trmm_patterns):
...     for substitution in match(expr, pattern):
...         print('{} with {}'.format(i, substitution))
0 with {A -> M3, B -> M2, t -> (), h -> ((M3)^T, M1)}
1 with {A -> M3, B -> M1, t -> (M3, M2), h -> ()}
2 with {A -> M3, B -> M1, t -> (M2), h -> ((M3)^T)}
```

As can be seen in the output, a total of three matches are found.

Design Considerations

There are plenty of implementations of syntactic matching and the algorithms are well known. Implementing pattern matching for MatchPy poses some challenges such as associativity and commutativity.

Associativity/Sequence variables

Associativity enables arbitrary grouping of arguments for matching: For example, 1 + a + b matches 1 + x_ with $\{x \mapsto a+b\}$ because we can group the arguments as 1 + (a + b). Basically, when regular variables are arguments of an associative function, they behave like sequence variables. Both can result in multiple distinct matches for a single pattern. In contrast, for syntactic patterns there is always at most one match. This means that the matching algorithm needs to be non-deterministic to explore all potential matches for associative terms or terms with sequence variables. We employ backtracking with the help of Python generators to enable this. Associative matching is NP-complete [BKN87].

Commutativity

Matching commutative terms is difficult because matches need to be found independent of the argument order. Commutative matching has been shown to be NP-complete, too [BKN87]. It is possible to find all matches by matching all permutations of the subjects arguments against all permutations of the pattern arguments. However, with this naive approach, a total of $n!m!$ combinations have to be matched where n is the number of subject arguments and m the number of pattern arguments. It is likely that most of these combinations do not match or yield redundant matches.

Instead, we interpret the arguments as a multiset, i.e. an orderless collection that allows repetition of elements. Also, we use the following order for matching the subterms of a commutative term:

1. Constant arguments
2. Matched variables, i.e. variables that already have a value assigned in the current substitution
3. Non-variable arguments
4. Repeat step 2
5. Regular variables
6. Sequence variables

Each of those steps reduces the search space for successive steps. This also means that if one step finds no match, the remaining steps do not have to be performed. Note that steps 3, 5 and 6 can yield multiple matches and backtracking is employed to check every combination. Since step 6 is the most involved, it is described in more detail in the next section.

Sequence Variables in Commutative Functions

The distribution of n subjects subterms onto m sequence variables within a commutative function symbol can yield up to m^n distinct solutions. Enumerating all of the solutions is accomplished by generating and solving several linear Diophantine equations. As an example, lets assume we want to match f(a, b, b, b) with f(x___, y__, y__) where f is commutative. This means that the possible distributions are given by the non-negative integer solutions of these equations:

$$
\begin{aligned}
1 &= x_a + 2y_a \\
3 &= x_b + 2y_b
\end{aligned}
$$

x_a determines how many times a is included in the substitution for x. Because y__ requires at least one term, we have the additional constraint $y_a + y_b \geq 1$. The only possible solution $x_a = x_b = y_b = 1 \wedge y_a = 0$ corresponds to the match substitution $\{x \mapsto (a,b), y \mapsto (b)\}$.

Extensive research has been done on solving linear Diophantine equations and linear Diophantine equation systems [Wei60], [Bon67], [Lam88], [CF89], [AHL00]. In our case the equations are actually independent expect for the additional constraints for plus variables. Also, the non-negative solutions can be found more easily. We use an adaptation of the algorithm used in SymPy which recursively reduces any linear Diophantine equation to equations of the form $ax + by = d$. Those can be solved efficiently with the Extended Euclidian algorithm [MVO96]. Then the solutions for those can be combined into a solution for the original equation.

All coefficients in those equations are likely very small since they correspond to the multiplicity of sequence variables. Similarly, the number of variables in the equations is usually small as they map to sequence variables. The constant is the multiplicity of a subject term and hence also usually small. Overall, the number of distinct equations that are solved is small and the solutions are cached. This reduces the impact of the sequence variables on the overall run time.

Optimizations

Since most applications for pattern matching repeatedly match a fixed set of patterns against multiple subjects, we implemented many-to-one matching for MatchPy. The goal of many-to-one matching is to utilize similarities between patterns to match them more efficiently. In this section, we give a brief overview of the many-to-one matching algorithm used by MatchPy. Full details can be found in the master thesis [Kre17b].

Many-to-one Matching

MatchPy includes two additional algorithms for matching: ManyToOneMatcher and DiscriminationNet. Both enable matching multiple patterns against a single subject much faster than matching each pattern individually using match. The latter can only be used for syntactic patterns and implements a state-of-the-art deterministic discrimination net. A discrimination net is a data structure similar to a decision tree or a finite automaton [Chr93], [Grä91], [NWE97]. The ManyToOneMatcher utilizes a generalized form of non-deterministic discrimination nets that support sequence variables and associative function symbols. Furthermore, as elaborated in the next section, it can also match commutative terms.

In Figure 1, an example for a non-deterministic discrimination net is shown. It contains three patterns that match Python lists:
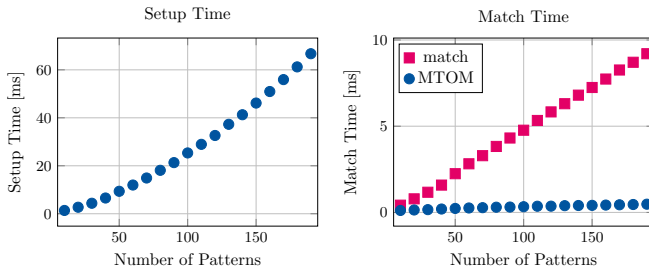
Fig. 1: Example Discrimination Net.

Fig. 2: Example for Order in Bipartite Graph.

One matches the list that consists of a single 1, the second one matches a list with exactly two elements where the last element is 0, and the third pattern matches any list where the first element is 1. Note, that these patterns can also match nested lists, e.g. the second pattern would also match `[[2, 1], 0]`.

Matching starts at the root and proceeds along the transitions. Simultaneously, the subject is traversed in preorder and each symbol is checked against the transitions. Only transitions matching the current subterm can be used. Once a final state is reached, its label gives a list of matching patterns. For non-deterministic discrimination nets, all possibilities need to be explored via backtracking. The discrimination net allows to reduce the matching costs, because common parts of different pattern only need to be matched once. For non-matching transitions, their whole subtree is pruned and all the patterns are excluded at once, further reducing the match cost.

In Figure 1, for the subject `[1, 0]`, there are two paths and therefore two matching patterns: `[y_, 0]` matches with $\{y \mapsto 1\}$ and `[1, x___]` matches with $\{x \mapsto 0\}$. Both the `y`-transition and the `1`-transition can be used in the second state to match a `1`.

Compared to existing discrimination net variants, we added transitions for the end of a compound term to support variadic functions. Furthermore, we added support for both associative function symbols and sequence variables. Finally, our discrimination net supports transitions restricted to symbol classes (i.e. `Symbol` subclasses) in addition to the ones that match just a specific symbol. We decided to use a non-deterministic discrimination net instead of a deterministic one, since the number of states of the later would grow exponentially with the number of patterns. While the `DiscriminationNet` also has support for sequence variables, in practice the net became to large to use with just a dozen patterns.

Commutative Many-to-one Matching

Many-to-one matching for commutative terms is more involved. We use a nested `CommutativeMatcher` which in turn uses another `ManyToOneMatcher` to match the subterms. Our approach is similar to the one used by Bachmair and Kirchner in their respective works [BCR+95], [KM01]. We match all the subterms of the commutative function in the subject with a many-to-one matcher constructed from the subpatterns of the commutative function in the pattern (except for sequence variables, which are handled separately). The resulting matches form a bipartite graph, where one set of nodes consists of the subject subterms and the other contains all the pattern subterms. Two nodes are connected by an edge iff the pattern matches the subject. Such an edge is also labeled with the match substitution(s). Finding an overall match is then accomplished by finding a maximum matching in this graph. However, for the matching to be valid, all the substitutions on its edges must be compatible, i.e. they cannot have contradicting replacements for the same variable. We use

the Hopcroft-Karp algorithm [HK73] to find an initial maximum matching. However, since we are also interested in all matches and the initial matching might have incompatible substitutions, we use the algorithm described by Uno, Fukuda and Matsui [FM94], [Uno97] to enumerate all maximum matchings.

To avoid yielding redundant matches, we extended the bipartite graph by introducing a total order over its two node sets. This enables determining whether the edges of a matching maintain the order induced by the subjects or whether some of the edges "cross". Formally, for all edge pairs $(p, s), (p', s') \in M$ we require $(s \equiv s' \wedge p > p') \implies s > s'$ to hold where M is the matching, s, s' are subjects, and p, p' are patterns. An example of this is given in Figure 2. The order of the nodes is indicated by the numbers next to them. The only two maximum matchings for this particular match graph are displayed. In the left matching, the edges with the same subject cross and hence this matching is discarded. The other matching is used because it maintains the order. This ensures that only unique matches are yielded. Once a matching for the subpatterns is obtained, the remaining subject arguments are distributed to sequence variables in the same way as for one-to-one matching.

Experiments

To evaluate the performance of MatchPy, we conducted experiments on an Intel Core i5-2500K 3.3 GHz CPU with 8GB of RAM. Our focus is on relative performance of one-to-one and many-to-one matching rather than the absolute performance.

Linear Algebra

The operations for the linear algebra problem are shown in Table 1. The patterns all match BLAS kernels similar to the example pattern which was previously described. The pattern set consists of 199 such patterns. Out of those, 61 have an addition as outermost operation, 135 are patterns for products, and 3 are patterns for single matrices. A lot of these patterns only differ in terms of constraints, e.g. there are ten distinct patterns matching $A \times B$ with different constraints on the two matrices. By removing the sequence variables from the product patterns, these pattern can be made syntactic when ignoring the multiplication's associativity. In the following, we refer to the set of patterns with sequence variables as `LinAlg` and the set of syntactic product patterns as `Syntactic`.

The subjects were randomly generated such that matrices had random properties and each factor could randomly be transposed/inverted. The number of factors was chosen according to a normal distribution with $\mu = 5$. The total subject set consisted of 70 random products and 30 random sums. Out of the pattern set, random subsets were used to examine the influence of the pattern set size on the matching time. Across multiple subsets and repetitions per subject, the mean match and setup times were measured. Matching was performed both with the `match`
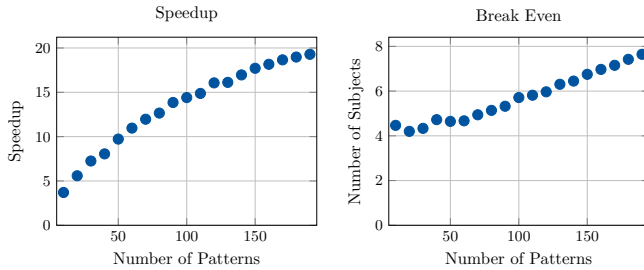
Fig. 3: *Timing Results for* `LinAlg`.

Fig. 4: *Comparison for* `LinAlg`.

function and the `ManyToOneMatcher` (MTOM). The results are displayed in Figure 3.

As expected, both setup and match times grow with the pattern set size. The growth of the many-to-one match time is much slower than the one for one-to-one matching. This is also expected since the simultaneous matching is more efficient. However, the growth of setup time for the many-to-one matcher beckons the question whether the speedup of the many-to-one matching is worth it.

Figure 4 depicts both the speedup and the break even point for many-to-one matching for `LinAlg`. The first graph indicates that the speedup of many-to-one matching increases with larger pattern sets. But in order to profit from that speedup, the setup cost of many-to-one matching must be amortized. Therefore, the second graph shows the break even point for many-to-one matching in terms of number of subjects. If for a given number of patterns and subjects the corresponding point is above the line, then many-to-one matching is overall faster. In this example, when matching more than eight times, many-to-one matching is overall always faster than one-to-one matching.

For the syntactic product patterns we compared the `match` function, the `ManyToOneMatcher` (MTOM) and the `DiscriminationNet` (DN). Again, randomly generated subjects were used. The resulting speedups and break even points are displayed in Figure 5.

In this case, the discrimination net is the fastest overall

Fig. 5: *Comparison for* `Syntactic`.

Fig. 6: *AST of the* `isinstance` *pattern.*

reaching a speedup of up to 60. However, because it also has the highest setup time, it only outperforms the many-to-one matcher after about 100 subjects for larger pattern set sizes. In practice, the discrimination net is likely the best choice for syntactic patterns, as long as the discrimination net does not grow to large. In the worst case, the size of the discrimination net can grow exponentially in the number of patterns.

Abstract Syntax Trees

Python includes a tool to convert code from Python 2 to Python 3. It is part of the standard library package `lib2to3` which has a collection of "fixers" that each convert one of the incompatible cases. To find matching parts of the code, those fixers use pattern matching on the abstract syntax tree (AST). Such an AST can be represented in the MatchPy data structures. We converted some of the patterns used by `lib2to3` both to demonstrate the generality of MatchPy and to evaluate the performance of many-to-one matching. Because the fixers are applied one after another and can modify the AST after each match, it would be difficult to use many-to-one matching for `lib2to3` in practice.

The following is an example of such a pattern:

```
power<
    'isinstance'
    trailer< '(' arglist< any ',' atom< '('
        args=testlist_gexp< any+ >
    ')' > > ')' >
>
```

It matches an `isinstance` expression with a tuple as second argument. Its tree structure is illustrated in Figure 6. The corresponding fixer cleans up duplications generated by previous fixers. For example `isinstance(x, (int, long))` would be converted by another fixer into `isinstance(x, (int, int))`, which in turn is then simplified to `isinstance(x, int)` by this fixer.

Out of the original 46 patterns, 36 could be converted to MatchPy patterns. Some patterns could not be converted, because they contain features that MatchPy does not support yet. The features include negated subpatterns (e.g. `not atom<'(' [any] ')'>`) and subpatterns that allow an aritrary number of repetitions (e.g. `any (',' any)+`).

Furthermore, some of the AST patterns contain alternative or optional subpatterns, e.g. `power<'input' args=trailer<'(' [any] ')'>>`. These features are also not directly supported by MatchPy, but they can be replicated by using multiple patterns. For those `lib2to3` patterns, all combinations of the alternatives were generated and added as invidual patterns. This resulted in about 1200 patterns for the many-to-one matcher that completely cover the original 36 patterns.

For the experiments, we used a file that combines the examples from the unittests of lib2to3 with about 900 non-empty lines. We compared the set of 36 patterns with the original matcher and the 1200 patterns with the many-to-one matcher. A total of about 560 matches are found. Overall, on average, our many-to-one matcher takes 0.7 seconds to find all matches, while the matcher from lib2to3 takes 1.8 seconds. This yields a speedup of approximately 2.5. However, the construction of the many-to-one matcher takes 1.4 seconds on average. However, this setup cost will be amortized by the faster matching for sufficiently large ASTs. The setup time can also mostly be eliminated by saving the many-to-one matcher to disk and loading it once required.

Compared to the one-to-one matching in MatchPy, the many-to-one matching achieves a speedup of about 60. This is due to the fact that for any given subject less than 1% of patterns match. By taking into account the setup time of the many-to-one matcher, the break even point for it is at about 200 subjects.

Conclusions

We have presented MatchPy, a pattern matching library for Python with support for sequence variables and associative/commutative functions. This library includes algorithms and data structures for both one-to-one and many-to-one matching. Because non-syntactic pattern matching is NP-hard, in the worst case the pattern matching times grows exponentially with the length of the pattern. Nonetheless, our experiments on real world examples indicate that many-to-one matching can give a significant speedup over one-to-one matching. However, the employed discrimination nets come with a one-time construction cost which needs to be amortized to benefit from their speedup. In our experiments, the break even point for many-to-one matching was always reached well within the typical number of subjects for the respective application. Therefore, many-to-one matching is likely to result in a compelling speedup in practice.

For syntactic patterns, we also compared the syntactic discrimination net with the many-to-one matcher. As expected, discrimination nets are faster at matching, but also have a significantly higher setup time. Furthermore, the number of states can grow exponentially with the number of patterns, making them unsuitable for some pattern sets. Overall, if applicable, discrimination nets offer better performance than a many-to-one matcher.

Which pattern matching algorithm is the fastest for a given application depends on many factors. Hence, it is not possible to give a general recommendation. Yet, the more subjects are matched against the same pattern set, the more likely it is that many-to-one outperforms one-to-one matching. In the experiments, a higher number of patterns lead to an increase of the speedup of many-to-one matching. In terms of the size of the many-to-one matcher, the growth of the net was sublinear in our experiments and still feasible for large pattern sets. The efficiency of using many-to-one matching also heavily depends on the actual pattern set, i.e. the degree of similarity and overlap between the patterns.

Future Work

We plan on extending MatchPy with more powerful pattern matching features to make it useful for an even wider range of applications. The greatest challenge with additional features is likely to implement them for many-to-one matching. In the following, we discuss some possibilities for extending the library.

Additional pattern features

In the future, we plan to implement similar functionality to the Repeated, Sequence, and Alternatives functions from Mathematica. These provide another level of expressive power which cannot be fully replicated with the current feature set of MatchPy. Another useful feature are context variables as described by Kutsia [Kut06]. They allow matching subterms at arbitrary depths which is especially useful for structures like XML. With context variables, MatchPy's pattern matching would be as powerful as XPath [RDS17] or CSS selectors [RJE17] for such structures. Similarly, function variables which can match a function symbol would also be useful for those applications.

Integration

Currently, in order to use MatchPy, existing data structures must be adapted to provide their children via an iterator. Where that is not possible, for example because the data structures are provided by a third party library, translation functions need to be applied. Also, some native data structures such as dicts are currently not supported directly. Therefore, it would be useful, to have a better way of using existing data structures with MatchPy.

In particular, easy integration with SymPy is an important goal, since it is a popular tool for working with symbolic mathematics. SymPy already implements a form of pattern matching which is less powerful than MatchPy. It lacks support for sequence variables, symbol wildcards and constraints. Each constant symbol in SymPy can have properties that allow it to be commutative or non-commutative. One benefit of this approach is easier modeling of linear algebra multiplication, where matrices and vectors do not commute, but scalars do. Better integration of MatchPy with SymPy would provide the users of SymPy with more powerful pattern matching tools. However, Matchpy would require selective commutativity to be fully compatible with SymPy. Also, SymPy supports older Python versions, while MatchPy requires Python 3.6.

Performance

If pattern matching is a major part of an application, its running time can significantly impact the overall speed. Reimplementing parts of MatchPy as a C module would likely result in a substantial speedup. Alternatively, adapting part of the code to Cython could be another option to increase the speed [BBS09], [WLØ09]. Furthermore, generating source code for a pattern set similar to parser generators for formal grammars could improve matching performance. While code generation for syntactic pattern matching has been the subject of various works [Aug85], [FM01], [Mar08], [MRV03], its application with the extended feature set of MatchPy is another potential area of future research. Also, additonal research on the viability of pattern matching with increasingly complex and large subjects or patterns is desirable. Parallelizing many-to-one matching is also a possibility to increase the overall speed which is worth exploring.

Functional pattern matching

Since Python does not have pattern matching as a language feature, MatchPy could be extended to provide a syntax similar to other functional programming languages. However, without a switch statement as part of the language, there is a limit to the syntax of this pattern expression. The following is an example of what such a syntax could look like:

```python
with match(f(a, b)):
    if case(f(x_, y_)):
        print("x={}, y={}".format(x, y)))
    elif case(f(z_)):
        ....
```

There are already several libraries for Python which implement such a functionality for syntactic patterns and native data structures (e.g. MacroPy [Hao14], patterns [Sch14] or PyPatt [Jen15]). However, the usefulness of this feature needs further evaluation.

REFERENCES

[AHL00]   Karen Aardal, Cor A. J. Hurkens, and Arjen K. Lenstra. Solving a system of linear diophantine equations with lower and upper bounds on the variables. *Mathematics of Operations Research*, 25(3):427–442, August 2000. doi:10.1287/moor.25.3.427.12219.

[Aug85]   Lennart Augustsson. Compiling pattern matching. In *Proc. of a Conference on Functional Programming Languages and Computer Architecture*, pages 368–381, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[BBS09]   Stefan Behnel, Robert W. Bradshaw, and Dag Sverre Seljebotn. Cython tutorial. In Gaël Varoquaux, Stéfan van der Walt, and Jarrod Millman, editors, *Proceedings of the 8th Python in Science Conference*, pages 4–14, Pasadena, CA USA, 2009.

[BCR93]   Leo Bachmair, Ta Chen, and I. V. Ramakrishnan. *Associative-commutative Discrimination Nets*, pages 61–74. 1993. doi:10.1007/3-540-56610-4_56.

[BCR+95]   Leo Bachmair, Ta Chen, I. V. Ramakrishnan, Siva Anantharaman, and Jacques Chabin. Experiments with associative-commutative discrimination nets. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'95, pages 348–354. Morgan Kaufmann, 1995.

[BKN87]   Dan Benanav, Deepak Kapur, and Paliath Narendran. Complexity of matching problems. *Journal of Symbolic Computation*, 3(1):203–216, February 1987. doi:10.1016/S0747-7171(87)80027-5.

[BN98]   Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, New York, NY, USA, 1998.

[Bon67]   James Bond. Calculating the general solution of a linear diophantine equation. *The American Mathematical Monthly*, 74(8):955, October 1967. doi:10.2307/2315274.

[BW84]   Alan Bundy and Lincoln Wallen. *Discrimination Net*, pages 31–31. 1984. doi:10.1007/978-3-642-96868-6_59.

[CF89]   Michael Clausen and Albrecht Fortenbacher. Efficient solution of linear diophantine equations. *Journal of Symbolic Computation*, 8(1-2):201–216, July 1989. doi:10.1016/s0747-7171(89)80025-2.

[Chr93]   Jim Christian. Flatterms, discrimination nets, and fast term rewriting. *Journal of Automated Reasoning*, 10(1):95–113, 1993. doi:10.1007/BF00881866.

[DCHD90]   J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990. doi:10.1145/77626.79170.

[DCHH88]   Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, March 1988. doi:10.1145/42288.42291.

[Eke95]   S. M. Eker. Associative-commutative matching via bipartite graph matching. *The Computer Journal*, 38(5):381–399, May 1995. doi:10.1093/comjnl/38.5.381.

[FM94]   K. Fukuda and T. Matsui. Finding all the perfect matchings in bipartite graphs. *Applied Mathematics Letters*, 7(1):15–18, January 1994. doi:10.1016/0893-9659(94)90045-0.

[FM01]   Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming - ICFP '01*. Association for Computing Machinery (ACM), 2001. doi:10.1145/507635.507641.

[Grä91]   Albert Gräf. Left-to-right tree pattern matching. In *Proceedings of the 4th International Conference on Rewriting Techniques and Applications*, RTA-91, pages 323–334, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[Hao14]   Li Haoyi. Macropy. https://github.com/lihaoyi/macropy, 2014. URL: https://github.com/lihaoyi/macropy.

[HK73]   John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, December 1973. doi:10.1137/0202019.

[Jen15]   Grant Jenks. Pypatt. https://pypi.python.org/pypi/pypatt, 2015. URL: https://pypi.python.org/pypi/pypatt.

[KL91]   E. Kounalis and D. Lugiez. Compilation of pattern matching with associative-commutative functions. In *TAPSOFT '91*, pages 57–73. Springer Nature, 1991. doi:10.1007/3-540-53982-4_4.

[KM01]   Hélène Kirchner and Pierre-Etienne Moreau. Promoting rewriting to a programming language: A compiler for non-deterministic rewrite programs in associative-commutative theories. *International Journal of Foundations of Computer Science*, 11(2):207–251, March 2001. doi:10.1142/S0129054101000412.

[Kre17a]   Manuel Krebber. Matchpy documentation. http://matchpy.readthedocs.io/, 06 2017. URL: http://matchpy.readthedocs.io/.

[Kre17b]   Manuel Krebber. Non-linear associative-commutative many-to-one pattern matching with sequence variables. Master's thesis, 2017. arXiv:1705.00907.

[Kut06]   Temur Kutsia. Context sequence matching for XML. *Electronic Notes in Theoretical Computer Science*, 157(2):47–65, May 2006. doi:10.1016/j.entcs.2005.12.045.

[Lam88]   J. L. Lambert. Finding a partial solution to a linear system of equations in positive integers. *Computers & Mathematics with Applications*, 15(3):209–212, 1988. doi:10.1016/0898-1221(88)90172-1.

[LHKK79]   C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979. doi:10.1145/355841.355847.

[LM94]   D. Lugiez and J. L. Moysset. Tree automata help one to solve equational formulae in AC-theories. *Journal of Symbolic Computation*, 18(4):297–318, October 1994. doi:10.1006/jsco.1994.1049.

[Mar08]   Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML - ML '08*. Association for Computing Machinery (ACM), 2008. doi:10.1145/1411304.1411311.

[MRV03]   Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. *A Pattern Matching Compiler for Multiple Target Languages*, pages 61–76. 2003. doi:10.1007/3-540-36579-6_5.

[MSP+17]   Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: Symbolic computing in python. *Peerj Computer Science*, 3:e103, January 2017. doi:10.7717/peerj-cs.103.

[MVO96]   Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[NWE97]   Nadia Nedjah, Colin D. Walter, and Stephen E. Eldridge. *Optimal Left-to-right Pattern-matching Automata*, pages 273–286. 1997. doi:10.1007/BFb0027016.

[Pö16]   Jan Pöschko. Mathics. http://mathics.github.io/, October 2016. URL: http://mathics.github.io/.

[RDS17]   Jonathan Robie, Michael Dyck, and Josh Spiegel. XML path language (XPath) 3.1. W3C recommendation, W3C, March 2017.

[RJE17]   Florian Rivoal, Tab Atkins Jr., and Elika Etemad. CSS snapshot 2017. W3C note, W3C, January 2017.

[Sch14]   Alexander Schepanovski. patterns. https://github.com/Suor/patterns, 2014. URL: https://github.com/Suor/patterns.

[Uno97]   Takeaki Uno. Algorithms for enumerating all perfect, maximum and maximal matchings in bipartite graphs. In *Algorithms and Computation*, pages 92–101. Springer Nature, 1997. doi:10.1007/3-540-63890-3_11.

[Wei60]   Robert Weinstock. Greatest common divisor of several integers and an associated linear diophantine equation. *The Amer-*

ican Mathematical Monthly, 67(7):664, August 1960. `doi:10.2307/2310105`.

[WLØ09] Ilmar M. Wilbers, Hans Petter Langtangen, and Åsmund Ødegård. Using cython to speed up numerical python programs. In *Proceedings of Mekit'09*, pages 495–512, 2009.

[WR16] Inc. Wolfram Research. Mathematica, 2016. URL: https://www.wolfram.com.