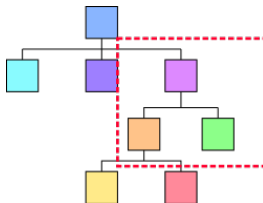


Tree matchings with Behavior Trees

FOSDEM 2019

February 2, 2019

How to recognize a complex subtree in a big tree



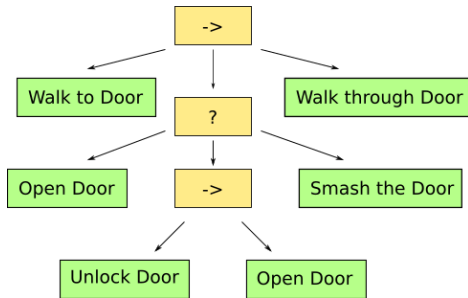
lionel@lse.epita.fr FOSDEM - Python Devroom 2019



- About Behavior Tree
- About Tree Matching...
- ...in python

A powerful abstraction to defined Process.

Common in Video Game to implement BOT AI.



3 components:

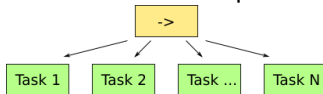
■ Task

Do a simple thing

Task n

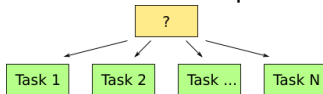
■ Sequence

Do the next if the previous succeed



■ Selector

Do the next if the previous failed



Task

```
from enum import IntEnum

Status = IntEnum('Status', [
    'Failure',
    'Success',
    'Running'
])

class Task:
    def tick(self, udata) -> Status:
        return Status.Success

class Tree:
    def __init__(self, *child):
        self.child = child
```

Sequence

```
class Sequence(Tree):  
    def tick(self, udata) -> Status:  
        for child in self.child:  
            childstatus = child.tick(udata)  
            if childstatus == Status.Running:  
                return Status.Running  
            elif childstatus == Status.Failure:  
                return Status.Failure  
        return Status.Success
```

Selector

```
class Selector(Tree):  
    def tick(self, udata) -> Status:  
        for child in self.child:  
            childstatus = child.tick(udata)  
            if childstatus == Status.Running:  
                return Status.Running  
            elif childstatus == Status.Success:  
                return Status.Success  
        return Status.Failure
```


Base on few principe, you could easlisy create your own abstraction

Concurrent

```
class Concurrent(Tree):
    def tick(self, udata) -> Status:
        for idx, child in enumerate(self.child):
            if self.status[idx] == Status.Running:
                self.status[idx] = child.tick(udata)
        if (not sum(map(lambda _: _ == Status.Running,
                        self.status))):
            if (sum(map(lambda _: _ == Status.Success,
                        self.status))):
                return Status.Success
            return Status.Failure
        return Status.Running
```

Use cases:

- Data Validation
- Data Transformation
- Data Generation

All in Compiler (AST)

Tree Handling:

- Descent recursive function (Tree Walk)
- Pattern Matching:
 - Identify nodes
 - Deconstruct it and do something
- Top-down

Top-down pattern matching is it enough?

Tree Reconstruction (update tree during walking) -> Bottom-Up

Intuition: Using BT to match subtree?

- Behavior Tree mimic matching subtree
- Versatile (Top-Down and/or Bottom-Up)

Get a generic python data tree

```
class A:
    def __init__(self, **kw):
        self.__dict__.update(kw)

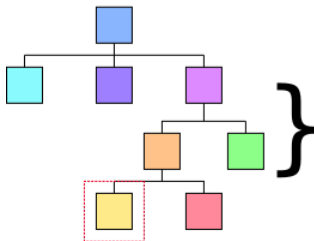
class B(dict):
    def __init__(self, d, **kw):
        self.update(d)
        self.__dict__.update(kw)

class C(list):
    def __init__(self, l, **kw):
        self.extend(l)
        self.__dict__.update(kw)
```

```
a = A(foo=42,  
      bar=C([  
          1, 2, 3,  
          B(h={'low': 1.2, 'high': 3.4})),  
          4, 5, 6  
      ],  
      meh=3  
    )  
)
```


Separation of concern:

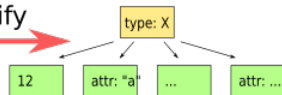
Data Tree



Walk

Behavior Tree

Notify



To handle it

- vars
- getattr
- collections.Mapping
- collections.Iterable
- yield

A generic walking function

```
def walk(tree):  
    if isinstance(tree, c.Mapping):  
        lsk = list(sorted(tree.keys()))  
        for k in lsk:  
            yield from walk(tree[k])  
        ...  
    elif (isinstance(tree, c.Iterable) \  
          and type(tree) not in {str, bytes}):  
        ls = enumerate(tree)  
        for idx, it in ls:  
            yield from walk(it)  
        ...  
    if hasattr(tree, '__dict__'):  
        attrs = vars(tree)  
        for attr in sorted(attrs.keys()):  
            yield from walk(attrs[attr])  
        ...
```

Notification

```
('value', tree)
('type', tree)
('attr', attr)
('key', k)
('idx', idx)
```

Behavior Tree Item:

- List: Sequence of value/idx
- Dict: Sequence of value/key
- Attributes: Sequence of value/attr
- Type: Concurrent Attributes/List/Dict

1 pattern = 1 BT

Don't store matching state IN the BT!

Module **treematching**

```
bt = AnyType(  
    List(  
        AnyIdx(Type(str, Value('lala'))),  
        AnyIdx(Type(int, Value(666))),  
    ),  
    Attrs(  
        Attr('foo', Type(int, AnyValue()))  
    )  
)  
e = MatchingBTree(bt)  
...  
match = e.match(tree)
```

Q/A!

- slides
- <https://github.com/LionelAuroux/treematching>