Week 8

Design of State Machines

Natalie Agus

Learning Objectives

- **Define** state machine (SM) as a function:
  - **output** function & **next state** function
- **Draw** state transition diagram & time-step table
- **Abstract** an SM using ABC module
- **Apply** BFS to perform state-space search
- Python concepts:
  - Explain **pure** functions
  - Explain Abstract Base Class (ABC)

Prerequisite

- **Inheritance**
  - Promote code reuse by **sharing** common functionalities
  - **Subclasses** can override or add new methods without changing **superclass**
- **vs Composition?**

```python
class Animal:
    def __init__(self, name: str) -> None:
        self.name: str = name

    def speak(self) -> str:
        return "Some generic sound"

class Dog(Animal):  # Dog inherits from Animal
    def speak(self) -> str:
        return "Woof!"

class Duck(Animal):  # Duck inherits from Animal
    def speak(self) -> str:
        return "Quack!"

# Usage
dog = Dog("Buddy")
duck = Duck("Daffy")

print(dog.name)     # Output: Buddy (inherited from Animal)
print(dog.speak())  # Output: Woof! (overridden in Dog)

print(duck.name)    # Output: Daffy (inherited from Animal)
print(duck.speak())
```

Prerequisite

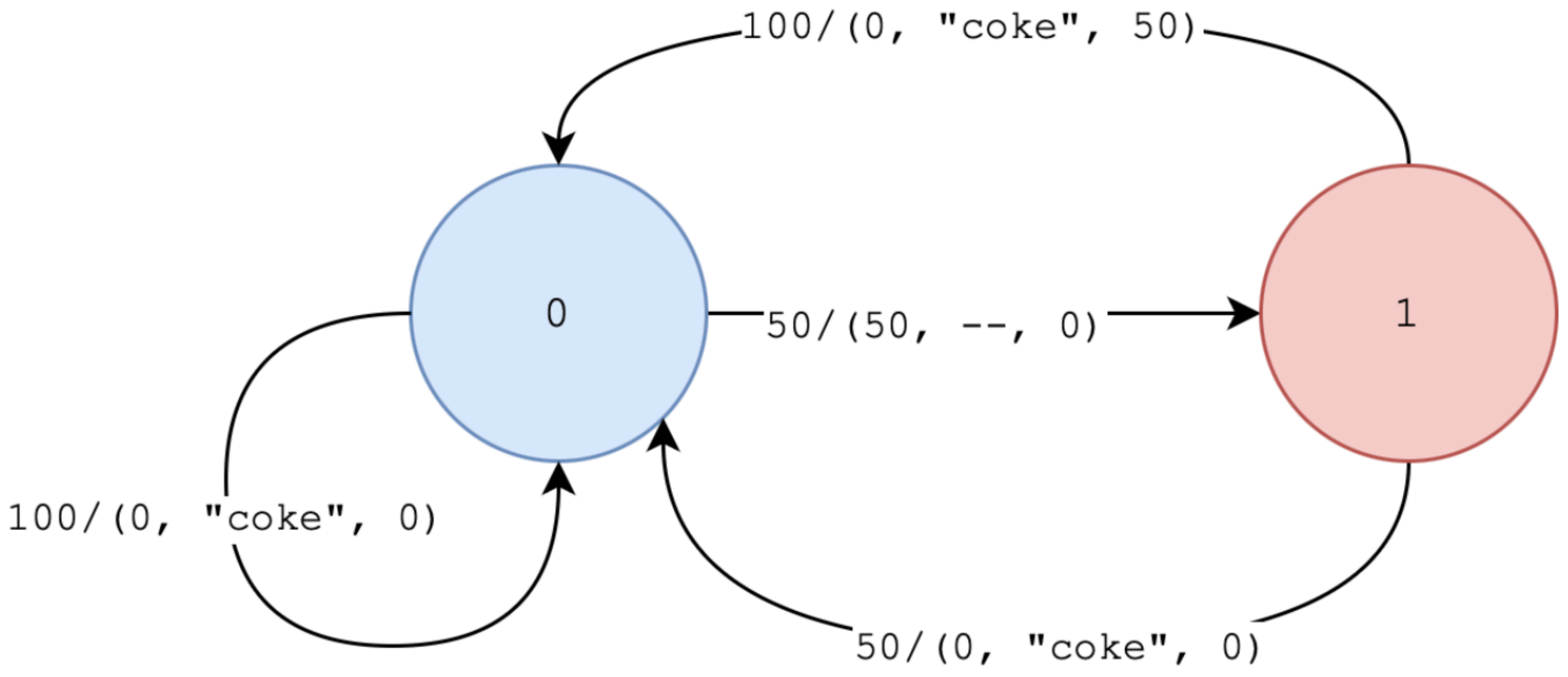
- **AttributeError**
  - Occurs when you try to access or assign an attribute that doesn't **exist** for a particular object

AttributeError: 'Person' object has no attribute 'age'

```python
class Person:
    def __init__(self, name) -> None:
        self.name: Any = name

p = Person("Alice")
print(p.age)  # Trying to access a non-existent attribute
```

```python
class Dog:
    def bark(self) -> None:
        print("Woof!")

d = Dog()
d.meow()  # Trying to call a non-existent method
```

State Machine Basics



- Initial State
- Arrow for **Transition**:
  - Input
  - Output
- Consider: **type** of input/ output

SM as a Function

$$o_t = f(i_t, i_{t-1}, i_{t-2}, \ldots)$$

$$o_t = f(i_t, s_t)$$

$$s_{t+1} = f(i_t, s_t)$$

- Output **depends** on **all** previous inputs
- Output is a **function** of all previous inputs

Function

What is a function?

- **A** mapping between a set of input (**domain**) and a set of output (**codomain**)
- **Mathematical vs Computer Science**
    - Block of code performing special task
    - Procedural
    - Flexible input & output: multiple input/output types at once
    - Has error handling

Pure Function

$$o_t = f(i_t)$$

- **Deterministic**: output always depends on current input only
- **No side effects**:
  - No changes outside of scope

```python

from typing import Union
from typing import List

def factorial(n: Union[int, float]) -> float:
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)


def square_numbers(numbers: List[int]) -> List[int]:
    return [n * n for n in numbers]
```

Impure Function

- **Non-deterministic**
  - Calling increment_counter() many times produces different output depending on counter's "*state*" (value)
- **Has** side effects
  - File is changed

```python
counter = 0  # Global variable

def increment_counter() -> int:
    global counter
    counter += 1
    return counter
```

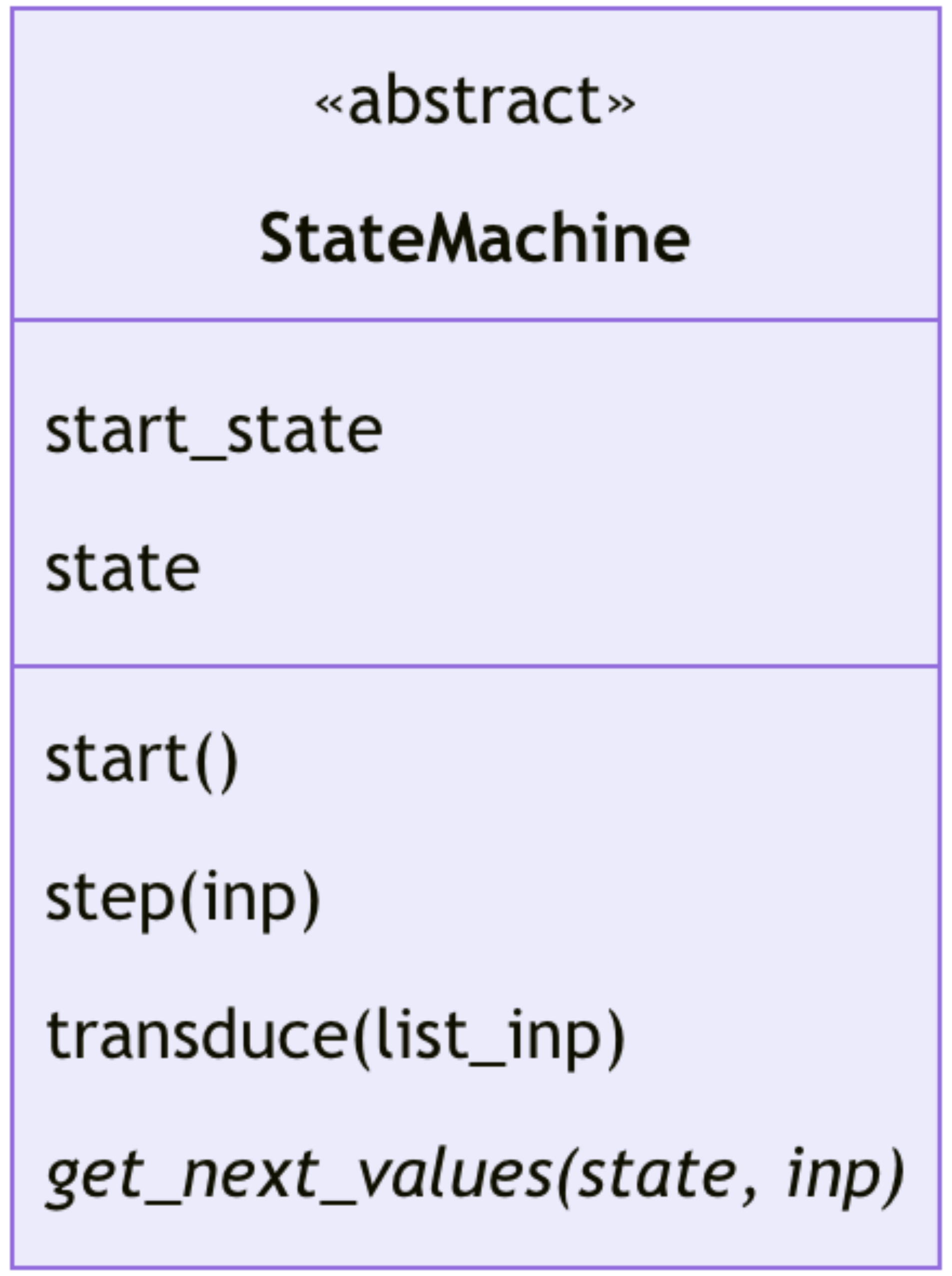
```python

# has side effect
def write_to_file(filename: str, text: str) -> None:
    with open(filename, 'a') as file:
        file.write(text + '\n')

```

Objects and State Machine

```python
class LightBox:
    def __init__(self) -> None:
        self.state = "off"

    def set_output(self, inp) -> str:
        if inp == 1 and self.state == "off":
            self.state = "on"
            return self.state
        if inp == 1 and self.state == "on":
            self.state = "off"
            return self.state
        return self.state

    def transduce(self, list_inp) -> None:
        for inp in list_inp:
            print(self.set_output(inp))

lb = LightBox()
lb.transduce([0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1])
```

- **Is this ideal?**
- What if you want to implement another SM: Coke Machine?
- Is there **boilerplate**?

Abstract Base Class for SM




```python
from abc import ABC, abstractmethod

class StateMachine(ABC):

    def start(self) -> None:
        self.state: None = self.start_state

    def step(self, inp) -> Any:
        ns: Any, o: Any = self.get_next_values(self.state, inp)
        self.state: Any = ns
        return o

    @property
    @abstractmethod
    def start_state(self) -> None:
        pass

    @abstractmethod
    def get_next_values(self, state, inp) -> None:
        pass

# cannot instantiate StateMachine class
# this will generate error

s = StateMachine()
```

- **@: decorator**
- A function that allows us to **modify** or **extend** the behavior of another function **without** changing it's actual code

Decorator

(extras)

```python
from typing import Union, Callable
Number = Union[int, float]

def log_decorator(func: Callable[..., Number]) -> Callable[..., Number]:
    def wrapper(*args: Number, **kwargs: Number) -> Number:
        # Check if all arguments are numbers
        if not all(isinstance(arg, (int, float)) for arg in args):
            raise TypeError("All arguments must be numbers.")
        print(f"Calling {func.__name__} with arguments {args} and {kwargs}")
        result: int | float = func(*args, **kwargs)
        print(f"{func.__name__} returned {result}")
        return result
    return wrapper

@log_decorator
def add(a: Number, b: Number) -> Number:
    return a + b
```

- **@property** and **@abstractmethod** are also functions implemented by Python under the hood
- They modify how the function below it **behaves**

Using the ABC

```python
from abc import ABC, abstractmethod

class StateMachine(ABC):

    def start(self) -> None:
        self.state: None = self.start_state

    def step(self, inp) -> Any:
        ns: Any, o: Any = self.get_next_values(self.state, inp)
        self.state: Any = ns
        return o

    @property
    @abstractmethod
    def start_state(self) -> None:
        pass

    @abstractmethod
    def get_next_values(self, state, inp) -> None:
        pass

# cannot instantiate StateMachine class
# this will generate error

s = StateMachine()
```

```python
class LightBoxSM(StateMachine):

    @property
    def start_state(self) -> Literal['off']:
        return "off"

    def get_next_values(self, state, inp) -> tuple[Any | Literal['on', 'off'], Liter…:

        if state == "off":
            if inp == 1:
                next_state = "on"
            else:
                next_state = "off"
        elif state == "on":
            if inp == 1:
                next_state = "off"
            else:
                next_state = "on"
        output: Literal['on'] | Literal['off'] = next_state
        return next_state, output

lb2 = LightBoxSM()
```

Designing a State Machine

This is an "art"

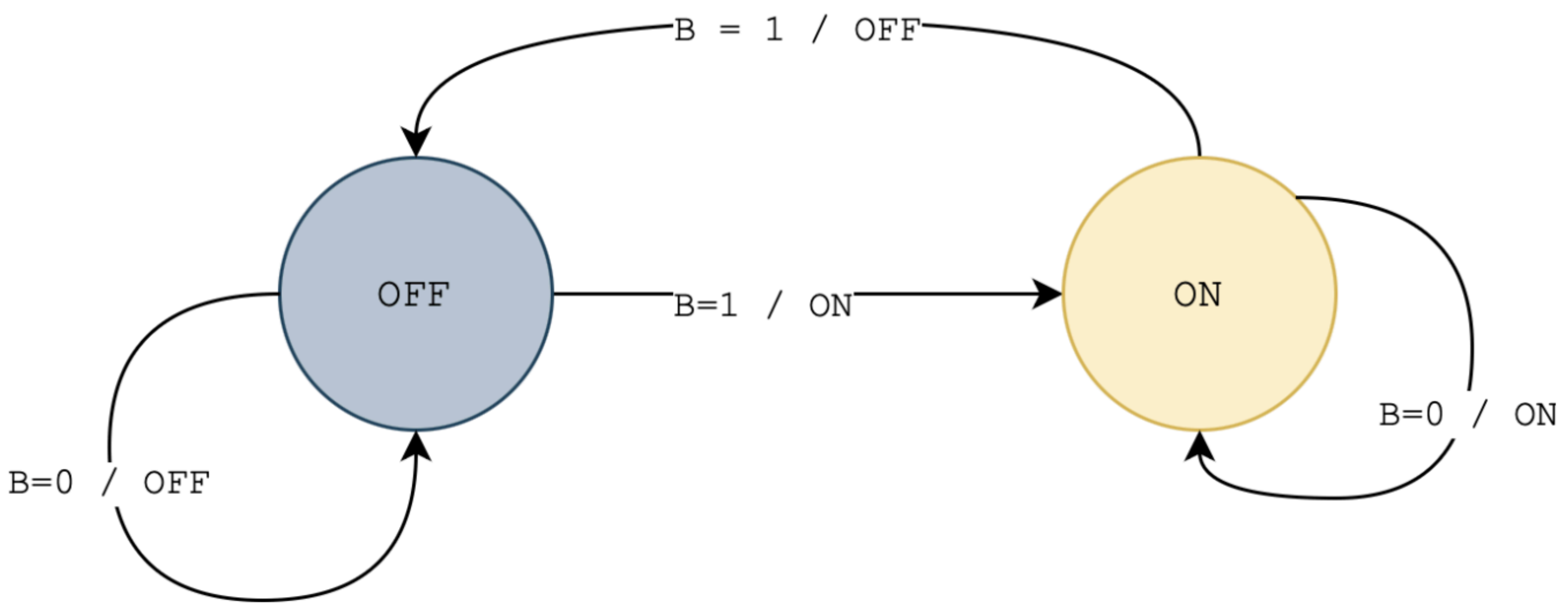
- **Is** it a state machine?
- Is the state **finite**?
- What are the **states** of the SM?
- What's the **starting state** of the SM?
- What are all possible **outputs**?
- What are all possible **inputs**?
- Determine the output and next state functions (**transition**)

Implementation

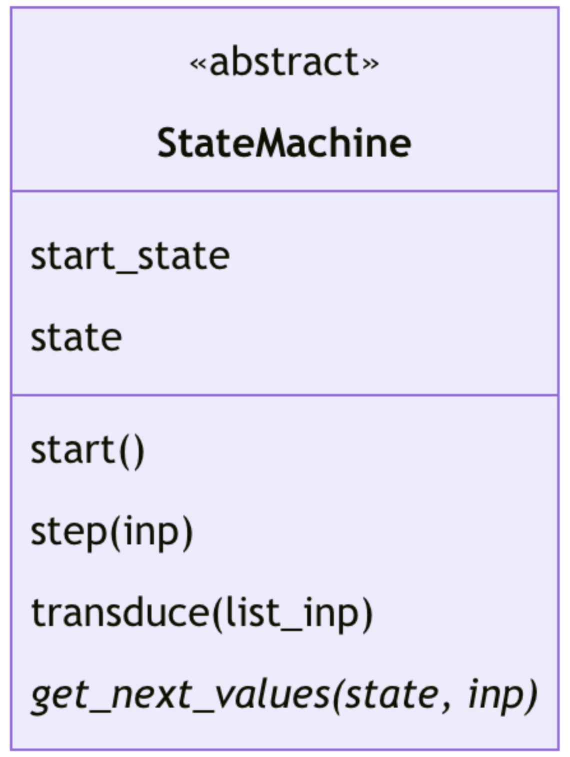
Using 10.020 SM ABC

- **Create** a new class inheriting StateMachine class
- Initialize the `start_state` attribute
- Implement `get_next_values()` function (**transition logic**)
  - A **pure** function
  - **No side effects!**
- *You can also create your own SM ABC depending on your application, but in 10.020 we stick to the above structure*

Example




```python
class LightBoxSM(StateMachine):

    @property
    def start_state(self):
        return "off"

    def get_next_values(self, state, inp):

        if state == "off":
            if inp == 1:
                next_state = "on"
            else:
                next_state = "off"
        elif state == "on":
            if inp == 1:
                next_state = "off"
            else:
                next_state = "on"
        output = next_state
        return next_state, output
```



- Which function calls `get_next_values`?
- Which function accesses `start_state`?

```python
lb2 = LightBoxSM()
lb2.start()
print(lb2.step(0))
```

The Time Step Table

	0	1	2	3	4
current state	0	10	35	30	41
current input	10	25	-5	11	-41
next state	10	35	30	41	0
current output	10	35	30	41	0

- Is it a state machine?
- What's the state of the SM?
- **Is the state finite?**
- What are all possible outputs?
- What are all possible inputs?
- Determine the output and next state functions (transition)

The Time Step Table

	0	1	2	3	4
current state	0	10	35	30	41
current input	10	25	-5	11	-41
next state	10	35	30	41	0
current output	10	35	30	41	0

```python
class AccumulatorSM(StateMachine):

    @property
    def start_state(self) -> Literal[0]:
        return 0

    def get_next_values(self, state, inp) -> tuple:
        output: Any = state + inp
        next_state: Any = output
        return next_state, output
acc = AccumulatorSM()
acc.start()
print(acc.step(10)) # outputs 10
print(acc.step(25)) # outputs 35
print(acc.step(-5)) # outputs 30
print(acc.step(11)) # outputs 41
print(acc.step(-41)) # outputs 0
```

The start() method

You can't use a machine you don't "start"

```python
acc = AccumulatorSM()
# acc.start()
print(acc.step(10)) # outputs 10
print(acc.step(25)) # outputs 35
print(acc.step(-5)) # outputs 30
print(acc.step(11)) # outputs 41
print(acc.step(-41)) # outputs 0
```

```
---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call last)

<ipython-input-29-234b6b850979> in <module>
      1 acc = AccumulatorSM()
      2 # acc.start()
----> 3 print(acc.step(10)) # outputs 10
      4 print(acc.step(25)) # outputs 35
      5 print(acc.step(-5)) # outputs 30


<ipython-input-22-556607471350> in step(self, inp)
      8     def step(self, inp):
      9         try:
---> 10             ns, o = self.get_next_values(self.state, inp)
     11         except ValueError:
     12             print("Did you return both next_state and output?")


AttributeError: 'AccumulatorSM' object has no attribute 'state'
```

The start() method

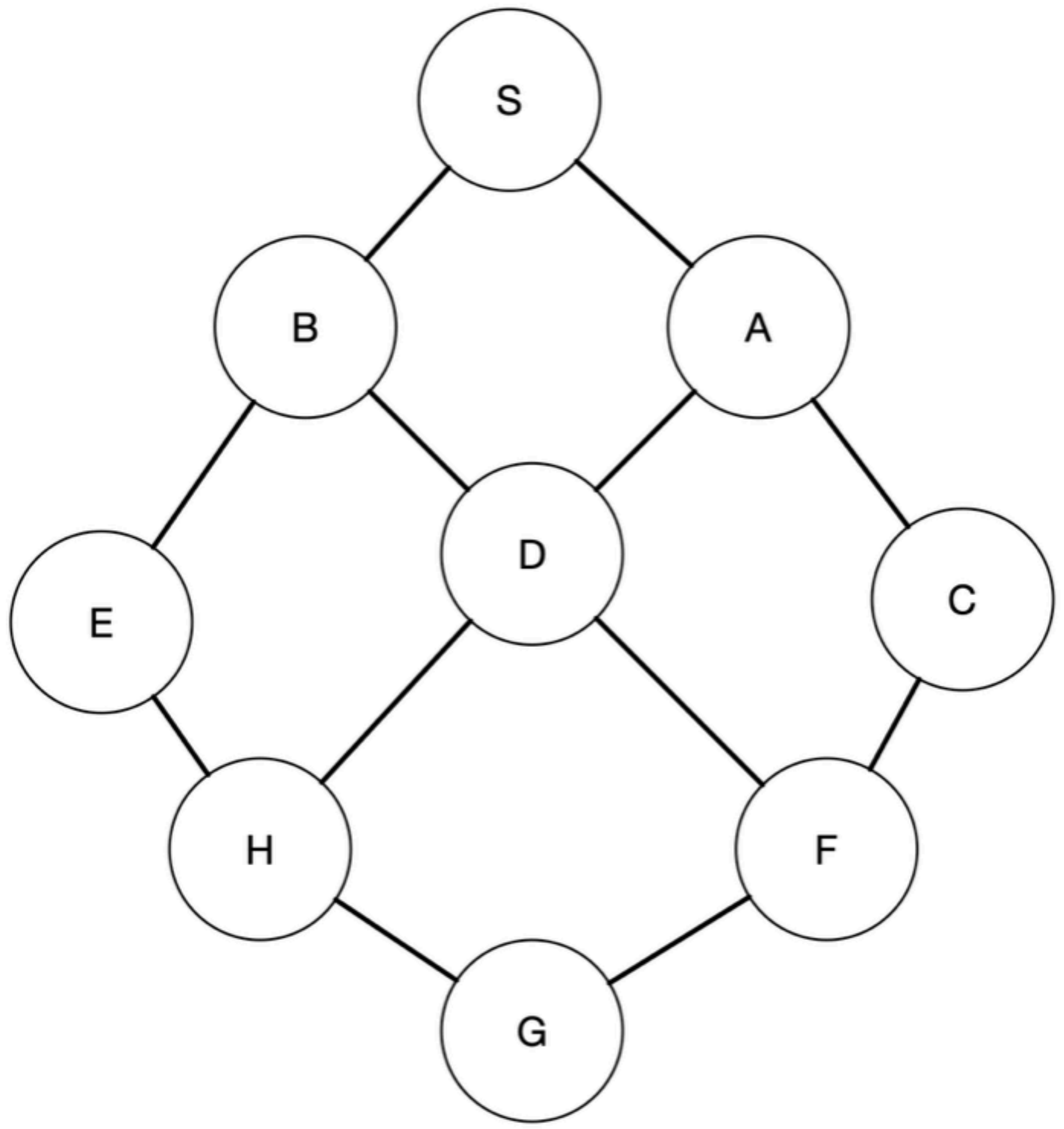
Useful for "restarting"

```python
class StateMachine(ABC):

    def start(self):
        self.state = self.start_state
```

```python
acc = AccumulatorSM()
acc.start()
print(acc.step(10)) # outputs 10
print(acc.step(25)) # outputs 35
print(acc.step(-5)) # outputs 30
print(acc.step(11)) # outputs 41
print(acc.step(-41)) # outputs 0

# restart machine
acc.start()
print(acc.step(100)) # outputs 100
print(acc.step(10)) # outputs 110
```

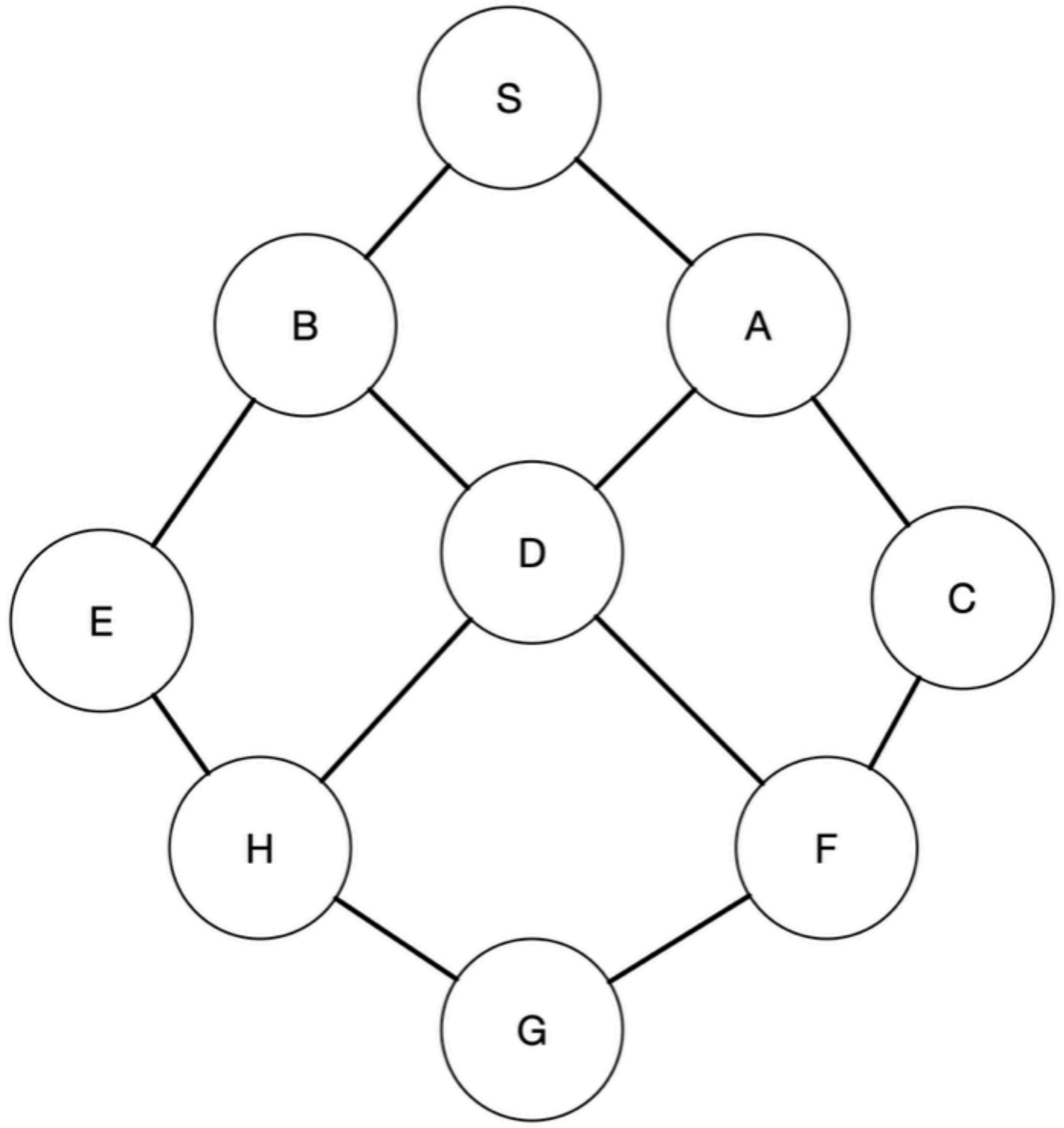
- Why can't we just set **self.state = self.start_state** in the `__init__`? Why do we need an extra `start()` function?

State Space Search



- **Goal** test
- **Legal** action list
- **Successor function**
- No arrow (bidirectional)
- Find a **path** from an initial state to a goal state within a defined set of possible states (the "state space")
  - Systematically explore **options**
  - Used commonly in AI and problem-solving programs

State Space Search



$$states = \{`S`, `A`, `B`, `C`, `D`, `E`, `F`, `G`, `H`\}$$

```
def goal_test(state):
    return state == 'G'
```

```
lambda state: state == 'G'
```

```
statemap = {'S': ['A', 'B'],
            'A': ['S', 'C', 'D'],
            'B': ['S', 'D', 'E'],
            'C': ['A', 'F'],
            'D': ['A', 'B', 'F', 'H'],
            'E': ['B', 'H'],
            'F': ['C', 'D', 'G'],
            'G': ['F', 'H'],
            'H': ['D', 'E', 'G']}
```

```
def statemap_successor(state, action):
    return statemap[state][action]
```

Lambdas*

Yes, programmers are lazy

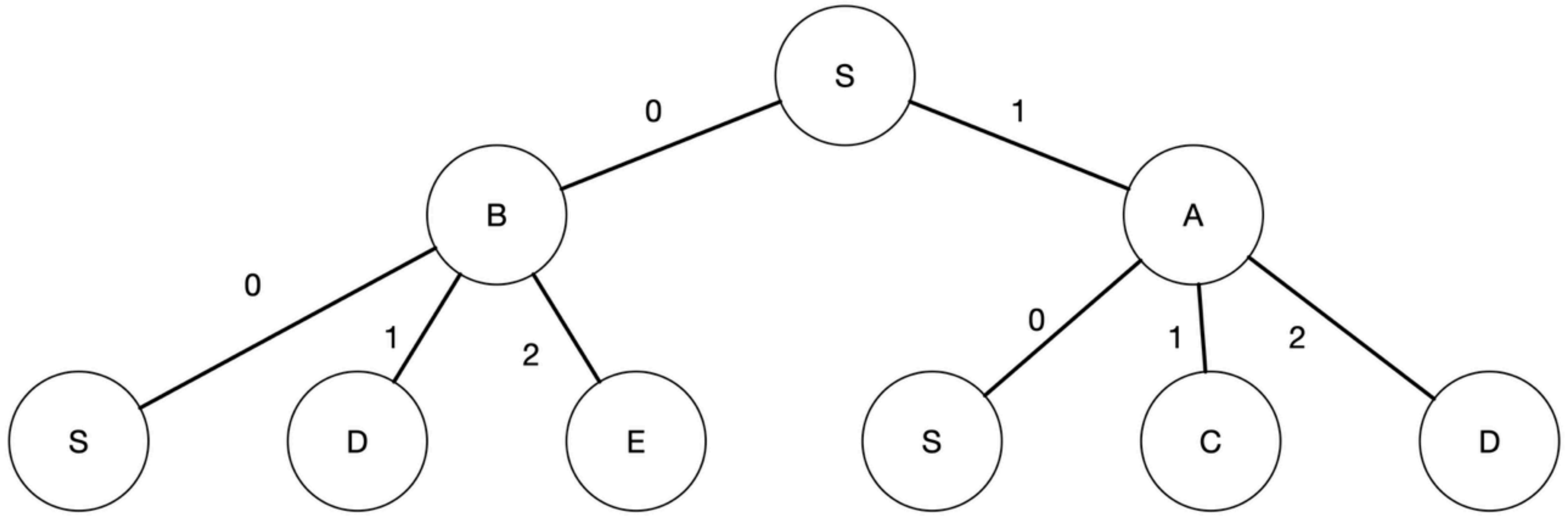
```
lambda arguments: expression
```

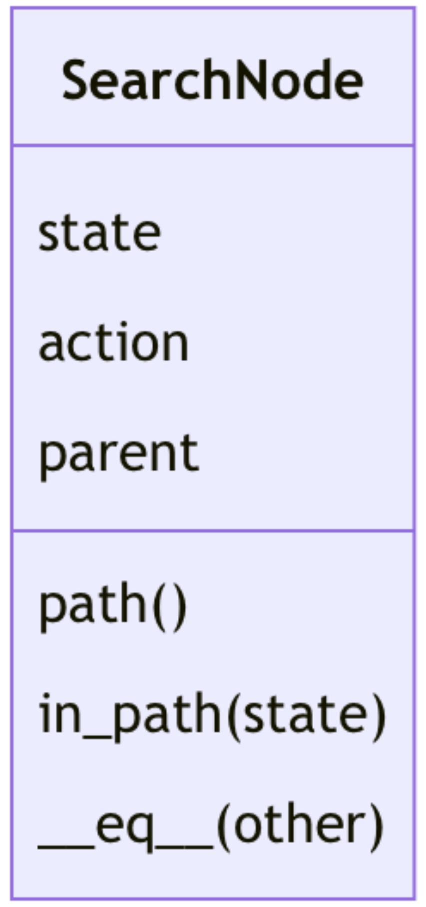
```
add: Callable[[int, int], int] = lambda x, y: x + y
print(add(2, 3))  # Output: 5


from typing import List

nums: List[int] = [1, 2, 3, 4]
squares: List[int] = list(map(lambda x: x ** 2, nums))
print(squares)  # Output: [1, 4, 9, 16]
```

- **Inline function**
- Limited to **single expression**
- **For simple**, short operation:
  - Quick, one-time usage
  - Don't need to write too much and give names

**Also known as anonymous functions/ closures, inline functions, block, or arrow functions in other programming languages*

Search Trees






```python
from abc import abstractmethod

class StateSpaceSearch(StateMachine):
    @property
    @abstractmethod
    def statemap(self):
        pass

    @property
    @abstractmethod
    def legal_inputs(self):
        pass

    @property
    @abstractmethod
    def start_state(self):
        return self.__start_state

    @start_state.setter
    @abstrctmethod
    def start_state(self, value):
        self.__start_state = value
```

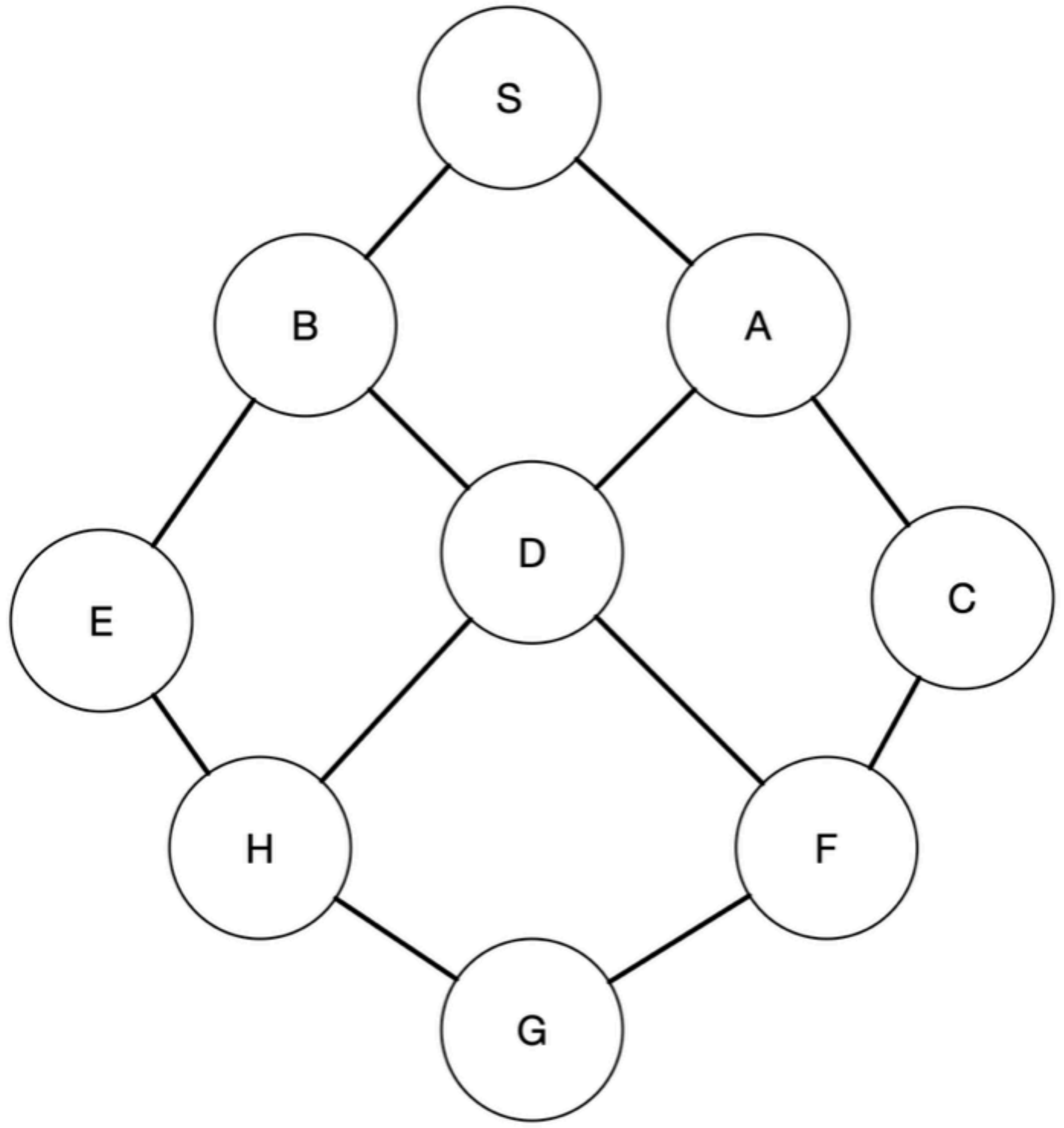
Search Trees

```python
class MapSM(StateSpaceSearch):

    def __init__(self, start):
        self.start_state = start

    @property
    def statemap(self):
        statemap = {"S": ["A", "B"],
                    "A": ["S", "C", "D"],
                    "B": ["S", "D", "E"],
                    "C": ["A", "F"],
                    "D": ["A", "B", "F", "H"],
                    "E": ["B", "H"],
                    "F": ["C", "D", "G"],
                    "H": ["D", "E", "G"],
                    "G": ["F", "H"]}
        return statemap
```

- **SearchNode** vs **StateSpaceSearch?**
    - Object vs the SM
- A **collection** of SearchNode forms the search tree

Search Trees

```
s: SearchNode = SearchNode(None, "S", None)
a:  SearchNode = SearchNode(0, "A", s)
b: SearchNode  = SearchNode(1, "B", s)
s1: SearchNode  = SearchNode(0, "S", a)
c: SearchNode  = SearchNode(1, "C", a)
d1: SearchNode  = SearchNode(2, "D", a)
s2: SearchNode  = SearchNode(0, "S", b)
d2: SearchNode  = SearchNode(1, "D", b)
e: SearchNode = SearchNode(2, "E", b)
a1: SearchNode  = SearchNode(0, "A", s1)
b1: SearchNode  = SearchNode(1, "B", s1)
a2: SearchNode  = SearchNode(0, "A", c)
f1: SearchNode  = SearchNode(1, "F", c)
a3: SearchNode  = SearchNode(0, "A", d1)
b2: SearchNode  = SearchNode(1, "B", d1)
f2: SearchNode  = SearchNode(2, "F", d1)
h1: SearchNode  = SearchNode(3, "H", d1)
a4: SearchNode  = SearchNode(0, "A", s2)
b3: SearchNode  = SearchNode(1, "B", s2)
a5: SearchNode  = SearchNode(0, "A", d2)
b4: SearchNode  = SearchNode(1, "B", d2)
f3: SearchNode  = SearchNode(2, "F", d2)
h2: SearchNode  = SearchNode(3, "H", d2)
b5: SearchNode  = SearchNode(0, "B", e)
h3: SearchNode  = SearchNode(1, "H", e)
```



Learning Objectives

- **Define** state machine (SM) as a function:
  - **output** function & **next state** function
- **Draw** state transition diagram & time-step table
- **Abstract** an SM using ABC module
- **Apply** BFS to perform state-space search
- Python concepts:
  - Explain **pure** functions
  - Explain Abstract Base Class (ABC)