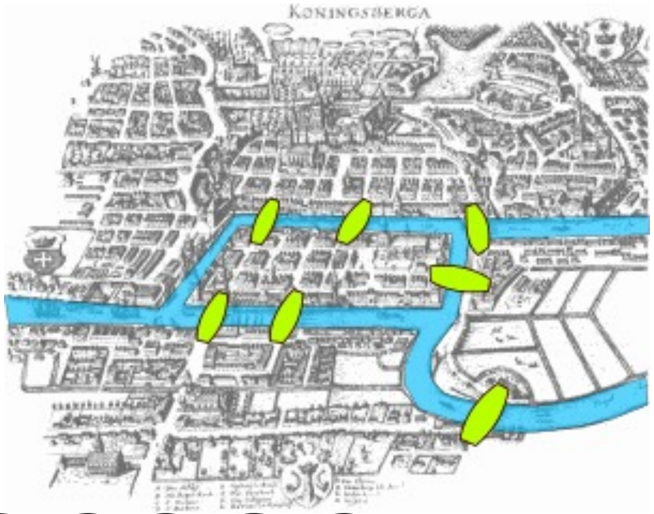


Implementing graphs with classes, Exceptions

(download slides and .py files to follow along)

Tim Kraska

MIT Department Of Electrical Engineering and Computer Science

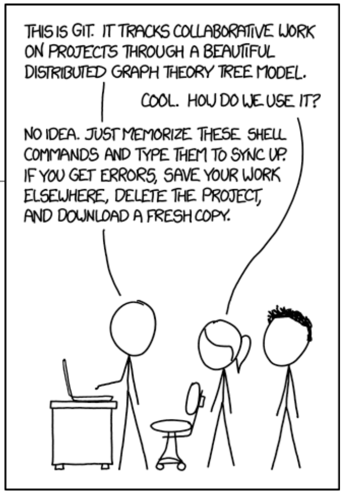
Announcements

- Pset 4 checkoff due Wed 9pm
- Pset 5 due Friday
- Midterm 2 next Wed 11/12
- Review session this Friday 11/7 during 7--9 pm.

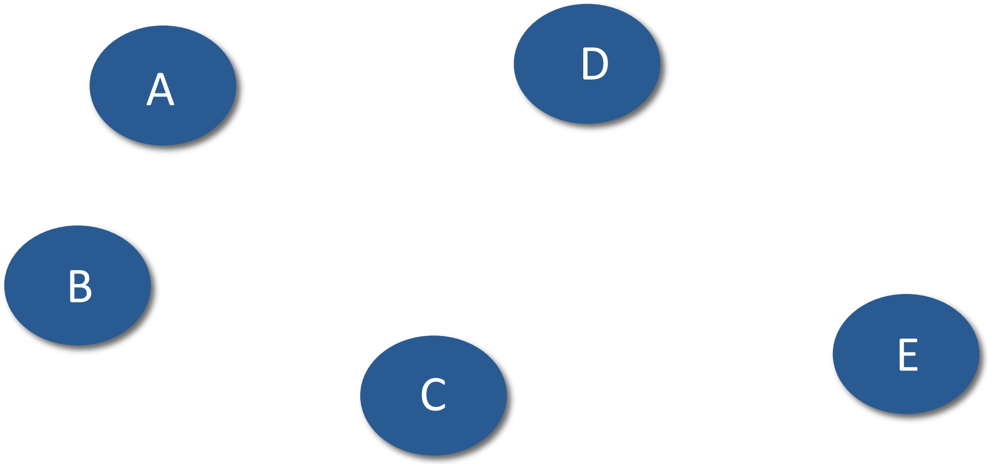
Topics

- Last week
  - Ideal Gas Law
  - Simulations with classes
- Today
  - Implementing graphs with classes
  - Exceptions
  - A*

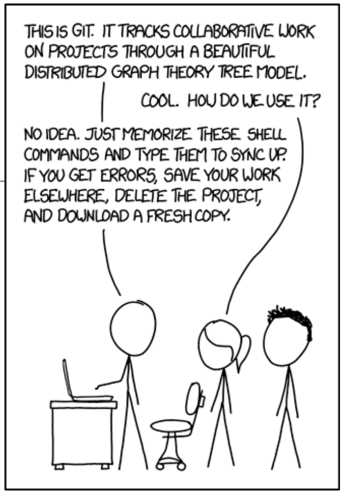
What is a Graph?



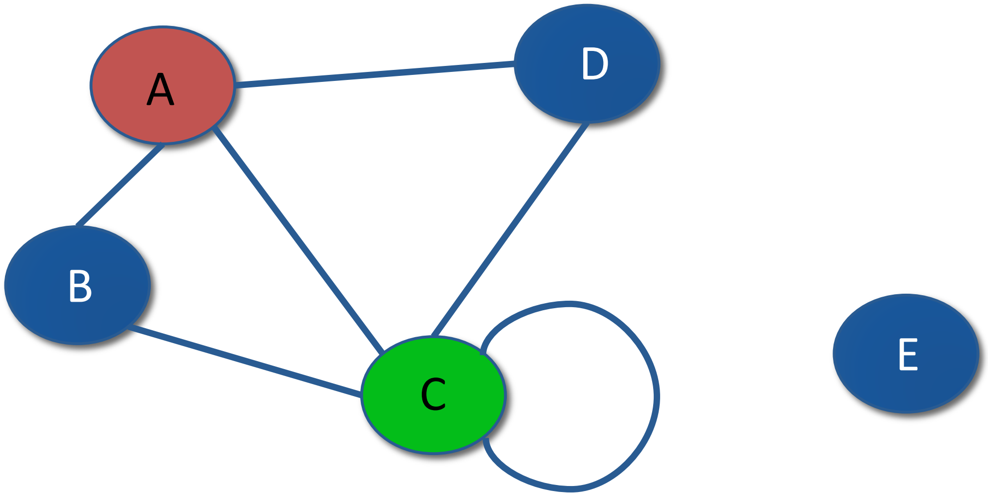
- Set of nodes (vertices)



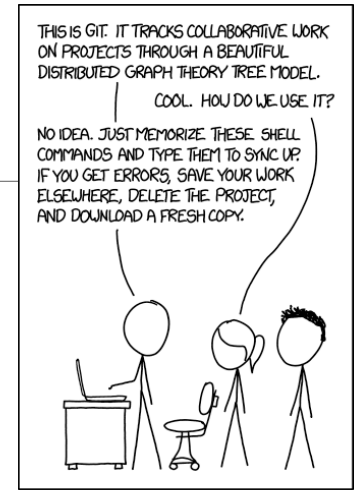
What is a Graph?



- Set of nodes (vertices)
  - Might have associated names or properties
- Set of edges (arcs) each connecting a pair of nodes
  - Undirected (graph)



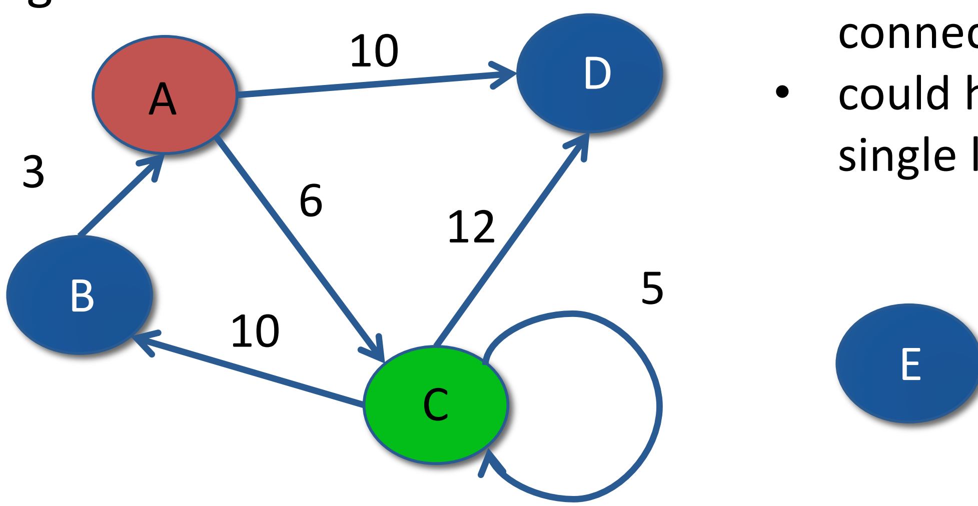
What is a Graph?



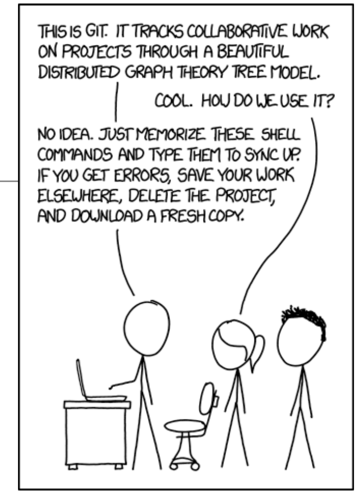
- Set of nodes (vertices)
  - Might have properties associated with them
- Set of edges (arcs) each connecting a pair of nodes
  - Undirected (graph)
  - Directed (digraph)
    - Source (parent) and destination (child) nodes
  - Unweighted or weighted
    - Assume non-negative

Graph:

- might not be completely connected
- could have loops, both single length and longer



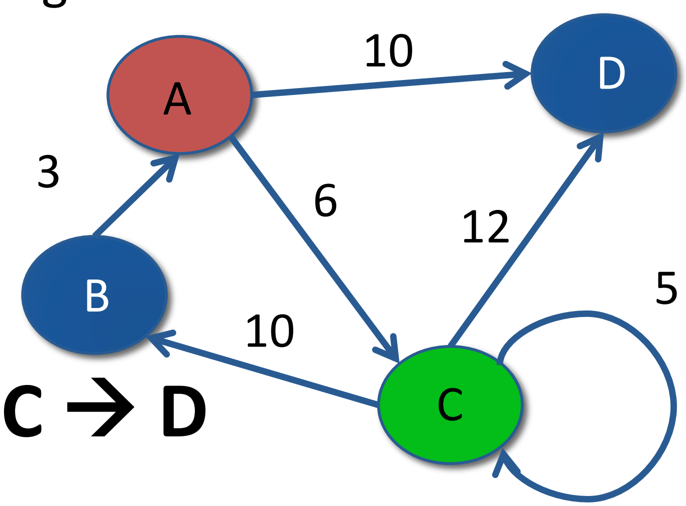
What is a Graph?



- Set of nodes (vertices)
  - Might have properties associated with them
- Set of edges (arcs) each connecting a pair of nodes
  - Undirected (graph)
  - Directed (digraph)
    - Source (parent) and destination (child) nodes
  - Unweighted or weighted
    - Assume non-negative

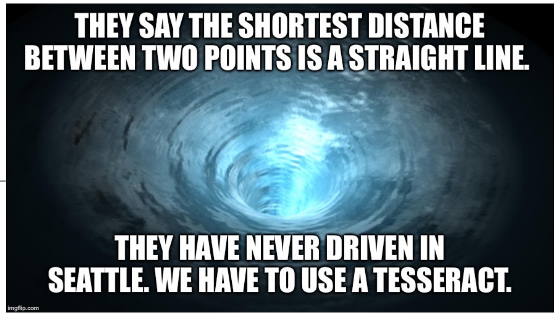
Graph:

- might not be completely connected
- could have loops, both single length and longer

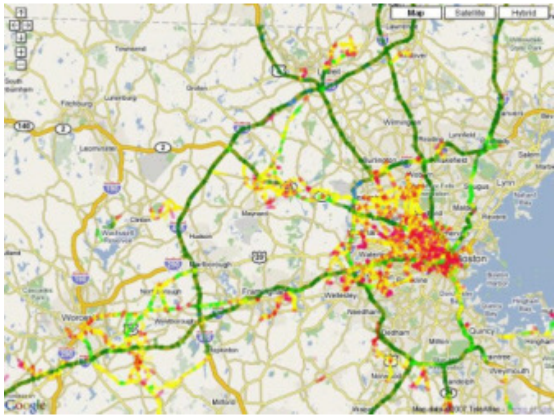


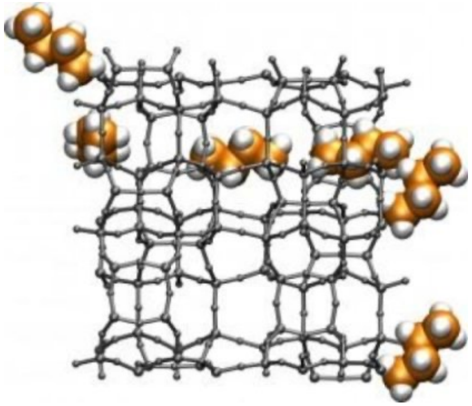
Path: B → A → C → D

Some Shortest Path Problems



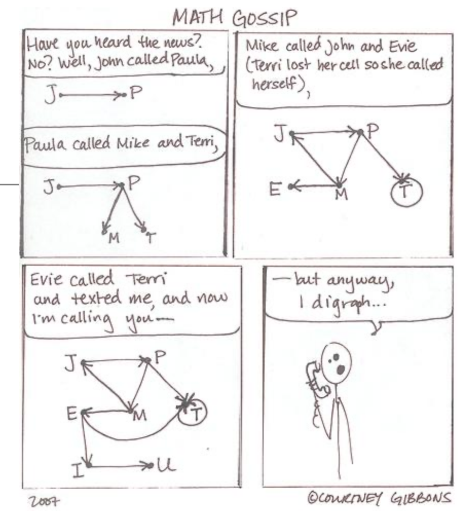
- Finding a route from one city to another
- Routing data on communication networks
- Warehouse logistics of storing and retrieving products
- Finding a path for a molecule through a chemical labyrinth

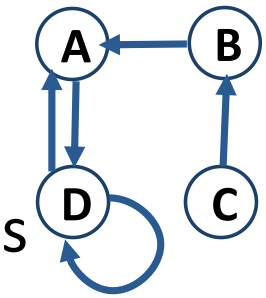




IMPLEMENTING GRAPHS

Representations of Digraphs





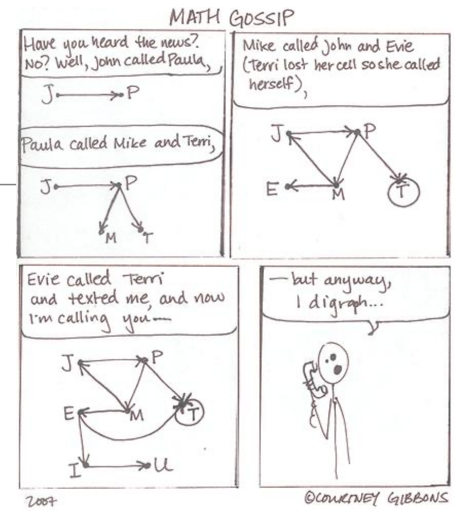
- **Digraph is a directed graph**
  - Edges pass in one direction only
  - Need to represent collection of edges
- **Adjacency matrix**
  - Rows: source nodes
  - Columns: destination nodes
  - Cell[s, d] = 1 if there is an edge from s to d
    = 0 otherwise
  - Note that in digraph, matrix is **not** symmetric
  - Uses **O(|nodes|**2) memory**
- **Assumes at most one arc between node pairs**
  - Easily generalized to multiple arcs with weights

d

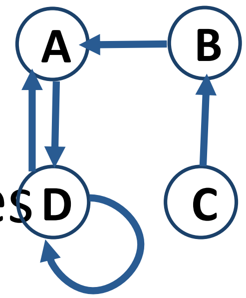
s

	A	B	C	D
A				1
B	1			
C		1		
D	1			1

Representations of Digraphs



- Digraph is a directed graph
  - Edges pass in one direction only
  - Need to represent collection of edges



- **Adjacency matrix**
  - Rows: source nodes
  - Columns: destination nodes
  - Cell[s, d] = 1 if there is an edge from s to d
    = 0 otherwise
  - Note that in digraph, matrix is **not** symmetric
  - Uses **O(|nodes|**2) memory**
- **Adjacency list**
  - Associate with each node a list of destination nodes that can be reached by one edge
  - Uses **O(|edges|) memory**, therefore good for **sparse** graphs

A: [D]
B: [A]
C: [B]
D: [A, D]

Class Exercise

Assume you want a data structure for the following requirements:

- Store general purpose directed and undirected graphs
- Model cities with their longitude and latitude for visualization and store main (undirected) connections between cities with their associated driving distance (km)
- Support different ways to calculate the distance between cities (Euclidian or great-circle distance)

What key classes and data structures would you use to implement the requirements?

Classes, part 1

```python
class Node:
    """Represents a generic graph node with only a name (string)."""

    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name

    def __eq__(self, other):
        return self.name == str(other)

    def __hash__(self):
        # Nodes are uniquely identified by their name
        return hash(self.name)
```

```python
class MapNode(Node):
    """Extends Node with (x, y) coordinate information."""

    def __init__(self, name, coords):
        super().__init__(name)
        if not (isinstance(coords, tuple) and len(coords) == 2):
            raise ValueError("coords must be a tuple (x, y)")
        self.coords = coords  # e.g., (-71.0589, 42.3601)

    def __str__(self):
        x, y = self.coords
        return f"{self.name} [{x:.4f}, {y:.4f}]"
```

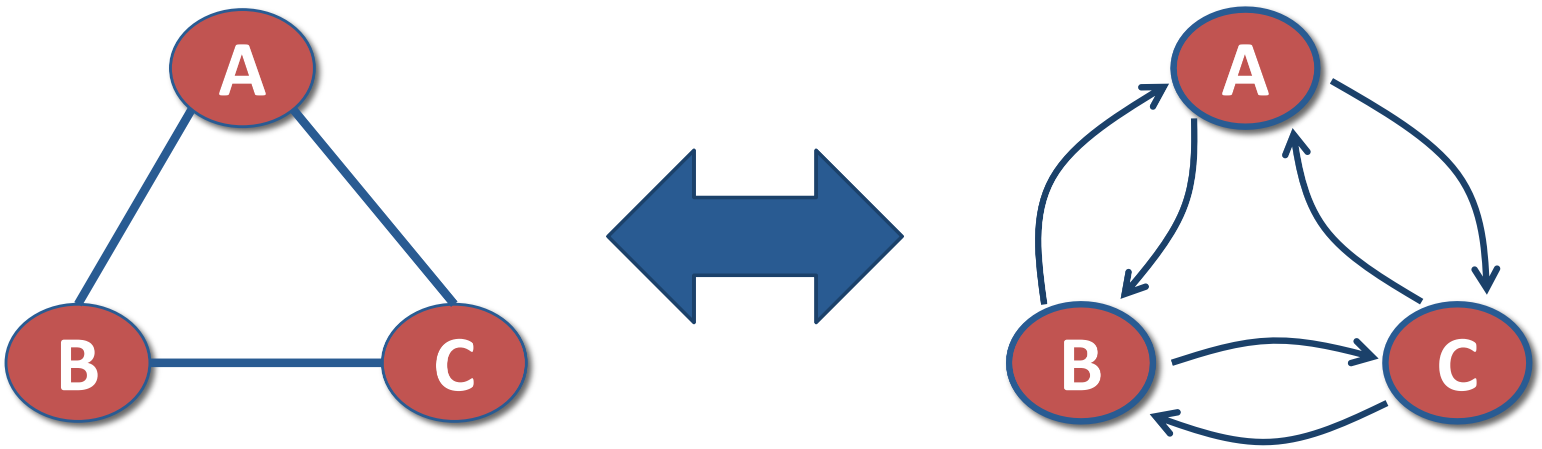
Classes, part 2

```python
class SimpleDigraph:
    """Represents a weighted directed graph using Nodes as keys."""

    def __init__(self, nodes=()):
        self._edges = {}  # dict: Node -> dict(Node -> weight)
        for node in nodes:
            self.add_node(node)

    def add_node(self, node):
        self._edges[node] = {}

    def get_node(self, id):
        if id in self._edges:
            return id

    def add_edge(self, src, dest, weight=1):
        """Add a directed edge between two Node objects (or names)."""
        src = self.get_node(src)
        dest = self.get_node(dest)
        self._edges[src][dest] = weight

    def get_all_nodes(self):
        return list(self._edges.keys())

    def outgoing_edges_of(self, node):
        return self._edges[node].copy()

    def children_of(self, node):
        return list(self._edges[node].keys())
```

```python
class SimpleGraph(SimpleDigraph):
    def add_edge(self, node1, node2, weight=1):
        super().add_edge(node1, node2, weight)
        super().add_edge(node2, node1, weight)
```

Implementing undirected graphs

- **Observation**: undirected edge can be represented as pair of opposite directed edges

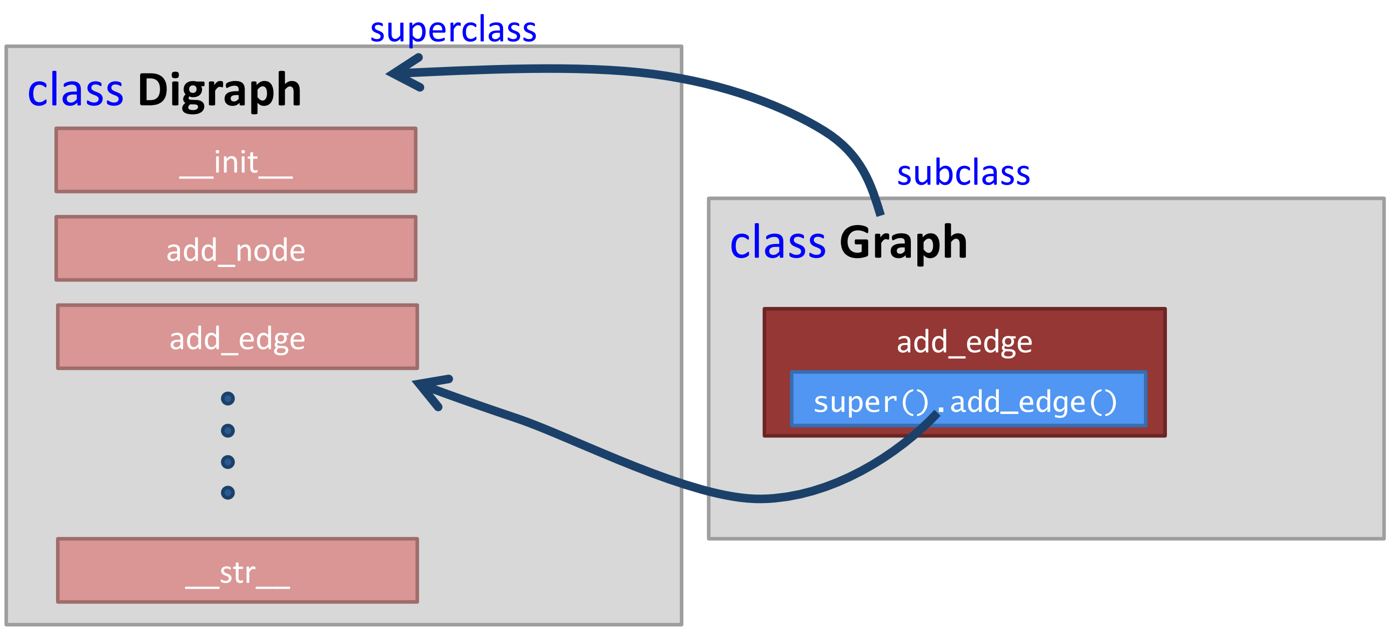


- **Insight**: Undirected graph can **reuse** most functionality from `class Digraph`
- Only have to respecify `add_edge()` method
  - Use **inheritance**, of course

Class Graph (undirected)

```python
class Graph(Digraph):
    """Represents an undirected graph with pairs
       of directed edges"""

    def add_edge(self, node1, node2, weight=1):
        super().add_edge(node1, node2, weight)
        super().add_edge(node2, node1, weight)
```

`super()` interprets *`self`* in the context of the parent class(es)



Classes, part 2

```python
class SimpleDigraph:
    """Represents a weighted directed graph using Nodes as keys."""

    def __init__(self, nodes=()):
        self._edges = {}  # dict: Node -> dict(Node -> weight)
        for node in nodes:
            self.add_node(node)

    def add_node(self, node):
        self._edges[node] = {}

    def get_node(self, id):
        if id in self._edges:
            return id

    def add_edge(self, src, dest, weight=1):
        """Add a directed edge between two Node objects (or names)."""
        src = self.get_node(src)
        dest = self.get_node(dest)
        self._edges[src][dest] = weight

    def get_all_nodes(self):
        return list(self._edges.keys())

    def outgoing_edges_of(self, node):
        return self._edges[node].copy()

    def children_of(self, node):
        return list(self._edges[node].keys())
```

```python
class SimpleGraph(SimpleDigraph):
    def add_edge(self, node1, node2, weight=1):
        super().add_edge(node1, node2, weight)
        super().add_edge(node2, node1, weight)
```

Why does Simple graph inherit from SimpleDigraph and not the other way around?

Class Graph (undirected)

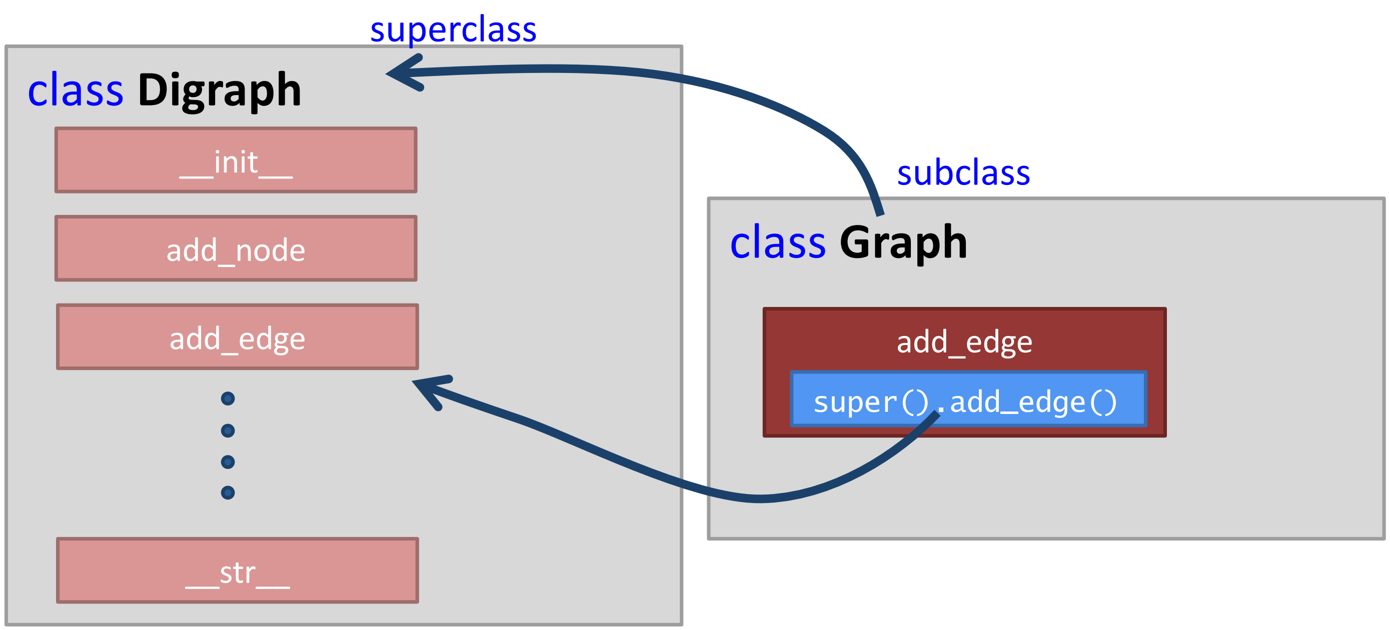
class **Graph** inherits
class **Digraph**'s functionality

Graph's
add_edge
overrides
Digraph's

```
class Graph(Digraph):
    """Represents an undirected graph with pairs
       of directed edges"""

    def add_edge(self, node1, node2, weight=1):
        super().add_edge(node1, node2, weight)
        super().add_edge(node2, node1, weight)
```

Add both directed edges using existing superclass method

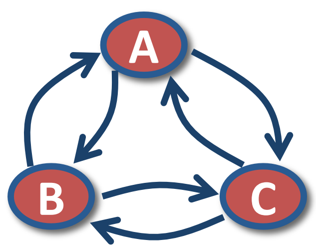
super() interprets *self* in the context of the parent class(es)



Designing Class Hierarchy

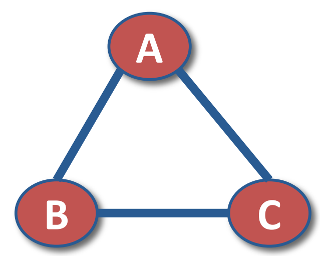
- Why make undirected **`Graph`** a subclass of **`Digraph`**, rather than vice versa?
  - Might seem like **`Digraph`** has additional "feature"
- Follow the **Substitution Principle**:
  - "**Subclass** behavior should be consistent with **superclass**"
- When **`Graph`** is subclass of **`Digraph`**:

Graph object interpreted as **`Digraph`** object



Connectivity:
A can reach B, C
B can reach A, C
C can reach A, B

Graph object



Connectivity:
A can reach B, C
B can reach A, C
C can reach A, B

Classes, part 2

```python
class SimpleDigraph:
    """Represents a weighted directed graph using Nodes as keys."""

    def __init__(self, nodes=()):
        self._edges = {}  # dict: Node -> dict(Node -> weight)
        for node in nodes:
            self.add_node(node)

    def add_node(self, node):
        self._edges[node] = {}

    def get_node(self, id):
        if id in self._edges:
            return id

    def add_edge(self, src, dest, weight=1):
        """Add a directed edge between two Node objects (or names)."""
        src = self.get_node(src)
        dest = self.get_node(dest)
        self._edges[src][dest] = weight

    def get_all_nodes(self):
        return list(self._edges.keys())

    def outgoing_edges_of(self, node):
        return self._edges[node].copy()

    def children_of(self, node):
        return list(self._edges[node].keys())
```

```python
class SimpleGraph(SimpleDigraph):
    def add_edge(self, node1, node2, weight=1):
        super().add_edge(node1, node2, weight)
        super().add_edge(node2, node1, weight)
```

How would you prevent that somebody adds something else than a Node to our graph

How would you allow to retrieve nodes using strings

How would you prevent that edges can be added for nodes that don’t exist?

Classes, part 3

```python
def add_node(self, node):
    """Add a Node object to the graph."""
    if not isinstance(node, Node):
        raise TypeError("add_node expects a Node instance")
    if node in self._edges:
        raise ValueError(f"Duplicate node: {node.name}")
    self._edges[node] = {}

def get_node(self, id):
    """Internal helper: resolve a node name or return node directly."""
    if isinstance(id, str) or isinstance(id, Node):
        for n in self._edges:
            if n.name == id:
                return n
        raise ValueError(f"Unknown node name: '{id}'")
    raise TypeError(f"Expected Node or str, got {type(id).__name__}")

def add_edge(self, src, dest, weight=1):
    """Add a directed edge between two Node objects (or names)."""
    src = self.get_node(src)
    dest = self.get_node(dest)
    self._edges[src][dest] = weight
```

EXCEPTIONS

(1) Exception Types
(2) Try-Except
(3) Raise
(4) Exceptions and Program Flow

Types of Problems with Code

- **Syntax**: program has **no meaning**, won't run

  fix syntax error (line number given)

today's lecture!

- **Crashes**: program has **meaning** but **invalid** at some point
  - converting string `'1'` to an integer is valid, but converting string `'abc'` to integer is an invalid operation

  exceptions & assertions

- **Returns wrong answer**: valid **meaning** throughout, not what you meant
  - we saw a lot of those examples in the mutability lecture

  debugging (Lecture 5)

- **Runs forever**: (likely) ditto

  debugging (Lecture 5)

Exception Types

Exception Types

- what happens when procedure execution hits an **unexpected condition**?
- get an **exception**... to what was expected
  - Dividing by zero → `ZeroDivisionError`
  - trying to access beyond list limits

```
test = [1, 7, 4]
test[4]
```
→ `IndexError`

  - operand does not have correct type

```
int([1,2,3])
```
→ `TypeError`

  - operand type ok, but value illegal

```
int('abc')
```
→ `ValueError`

(note that this is different from the list example, `list` can never be converted to `int` but some `string` can be converted to `int`, e.g., int('1'))

Exception Types

- referencing a non-existing variable, local or global name not found

    `a` → `NameError`

- IO systems reports malfunction

    file not found → `IOError`

Python Exception Hierarchy

- most built-in exceptions are **errors**, some are **warnings**
  - https://docs.python.org/3/library/exceptions.html#exception-hierarchy

```
BaseException
 ├── BaseExceptionGroup
 ├── GeneratorExit
 ├── KeyboardInterrupt
 ├── SystemExit
 └── Exception
      ├── ArithmeticError
      │    ├── FloatingPointError
      │    ├── OverflowError
      │    └── ZeroDivisionError
      ├── AssertionError
      ├── AttributeError
      ├── BufferError
      ├── EOFError
      ├── ExceptionGroup [BaseExceptionGroup]
      ├── ImportError
      │    └── ModuleNotFoundError
      ├── LookupError
      │    ├── IndexError
      │    └── KeyError
      ├── MemoryError
      ├── NameError
      │    └── UnboundLocalError
      ├── OSError
      │    ├── BlockingIOError
```

can define your own exceptions, based on Python's class system (discussed next week)

Try-Except

How to handle Exceptions: Try-Except

- **Problem:** Normally the program just stops/crashes when they encounter a problem like this. But that isn't a good strategy.
- **Better:** **recognize** when those things happen and have an **alternate plan** in place.
- **Solution:** wrap code into **try-except** block.

- **Example:**
  - In many programs, users provide some sort of input.
  - If they provide invalid inputs (e.g., letters instead of numbers), we want to gracefully handle this instead of letting our program crash.

Try-Except

- when code in **try** fails, **except** block is executed.

```
try:
    a = int(input("Tell me one number: "))
    b = int(input("Tell me another number: "))
    print("a + b =", a + b)
    print("a / b =", a / b)
except:
    print("Bug in user input.")

print("Continue execution from here.")
```

Benefit: Code does not crash even with invalid input! Code execution continues.

```
Tell me one number: a
Bug in user input.
Continue execution from here.
```

Try-Except

- depending on **exception type** you can **execute different code.**

```
try:
    a = int(input("Tell me one number: "))
    b = int(input("Tell me another number: "))
    print("a + b =", a + b)
    print("a / b =", a / b)
except ValueError:
    print("Could not convert to a number.")
except ZeroDivisionError:
    print("Can't divide by zero")
except KeyboardInterrupt:
    # Press Ctrl-C to generate a KeyboardInterrupt exception
    print("User Interrupted Program")
except:
    print("Something went very wrong.")
```

Try-Except-Else-Finally:

```
try:
    a = int(input("Tell me one number: "))
    b = int(input("Tell me another number: "))
    print("a + b =", a + b)
    print("a / b =", a / b)
except ValueError:
    print("Could not convert to a number.")
except ZeroDivisionError:
    print("Can't divide by zero")
except:
    print("Something went very wrong.")
else:
```

only runs if **no** except block was executed

```
    print("Everything Worked Out Fine!")
finally:
```

always runs (can be used to e.g., close open connections (files, sockets))

```
    print("This is executed independent of success or failure.")
```

Try-Except: Error Object

```
try:
    a = int(input("Tell me one number: "))
    b = int(input("Tell me another number: "))
    print("a + b =", a + b)
    print("a / b =", a / b)
except ValueError as error:
    print("Returned Error:", error)
    print("Could not convert to a number.")
except ZeroDivisionError as error:
    print("Returned Error:", error)
    print("Can't divide by zero")

print("Continue execution from here.")
```

binds the variable name 'error' to the ValueError object

```
Tell me one number: 5
Tell me another number: 0
a + b = 5
Returned Error: division by zero
Can't divide by zero
Continue execution from here.
```

Try-Except: Example Jogging Speeds

- Handling zero division

```python
def collect_ratios_2(numers, denoms, collection):
    for i in range(len(numers)):
        try:
            ratio = numers[i] / denoms[i]
            collection.append(ratio)
        except ZeroDivisionError:
            print("Handling zero division")
            collection.append(float('nan')) # nan = not a number
```

```python
distances = [2, 5, 4]
times = [16, 0, 40]
speed_history = [6.5, 5.7]
collect_ratios_2(distances, times, speed_history)
print(f"{speed_history = }")
```

Try-Except: Example Jogging Speeds

- Handling different lengths of distance and time lists

```
def collect_ratios_3(numers, denoms, collection):
    for i in range(len(numers)):
        try:
            ratio = numers[i] / denoms[i]
            collection.append(ratio)
        except ZeroDivisionError:
            print("Handling zero division")
            collection.append(float('nan'))
        except IndexError:
            print("Handling index out of range")
```

```
distances = [2, 5, 4]
times = [16, 0]        Last entry missing
speed_history = [6.5, 5.7]
collect_ratios_3(distances, times, speed_history)
print(f"{speed_history = }")
```

Exceptions and Program Flow

Exceptions Augment the Program Flow

- program is a stack of nested function calls
- exceptions unwind the stack to the nearest applicable exception handler
  - if there is **no matching except block** on the lowest function call level where the exception occurs, the program looks on the **next higher level** on the stack for a matching except block
  - if it can't find one on the top of the stack, the exception is thrown back to the user on the console
- example: TypeError with Tuple from Jogging Speed Example

```
distances = [2, 5, 4]
times = [16, 7, 10]
speed_history = (6.5, 5.7)
```

Exceptions Augment the Program Flow

```python
def collect_ratios_4(distances, times, speed_history):
    """
    Receive jogging distances in miles, times in minutes.
    Calculate and append mph average speeds to speed history list.
    """
    #convert minutes into times per hour
    times_hr = []
    for t in times:
        times_hr.append(t / 60)

    #compute miles per hour and append to speed history list
    for i in range(len(distances)):
        try:
            ratio = distances[i] / times_hr[i]
            speed_history.append(ratio)
        except ZeroDivisionError:
            print("Handling zero division")
            speed_history.append(float('nan'))
```

```python
distances = [2, 5, 4]
times = [16, 7, 10]
speed_history = (6.5, 5.7)
try:
    collect_ratios_4(distances, times, speed_history)
except AttributeError as error:
    print("Returned Error:", error)
print(f"{speed_history = }")
```

Example: Different Exceptions Jogging Speed Code

- `ZeroDivisonError` from having speeds that are 0 is **handled locally** in collect_ratios()

- `AttributeError` from trying to append to a tuple is **handled on the top of the stack** because no matching exception is found on the lower levels of the call stack

```
<global frame>
     .....
     .....
     try:
          collect_ratios(....)
     except AttributeError:
          .......
```

```
<collect_ratios>
     for i in ...
     try: ....
     except ZeroDivisionError:
     .....
```

Raise

Raising our own Exceptions

- **raising our own exception** allows us to shadow the current exception with our own
- can give **more specific information** to the user what went wrong because we know more about our function's purpose

```
raise <exceptionName>(<arguments>)
raise ValueError("something is wrong")
```

keyword

name of error you want to raise

optional, but typically a string with a message

Raising Exceptions: Jogging Speeds Example Continued

- raise exception for unrealistic jogging speeds
- Python would normally not create an exception for this

```
#compute miles per hour and append to speed history list
for i in range(len(distances)):
    ratio = distances[i] / times_hr[i]
    if ratio > 30: raise ValueError(f"Jogging Speed is unrealistically 
high with {ratio} miles per hour.")
    speed_history.append(ratio)
```

```
distances = [10, 5, 4]
times = [16, 0]
speed_history = [6.5, 5.7]
try:
    collect_ratios_5(distances, times, speed_history)
except ValueError as error:
    print("Returned Error:", error)
print(f"{speed_history = }")
```

Assertions

Assertions vs. Exceptions

- similar but different:
  - **exception** is for something **outside of your responsibility**
    - **signals to the user** that your program isn't designed to handle their input
    - are designed to handle errors, give user a useful message
  - **assertion** is something **your code design should guarantee**
    - assertions are a special type of Exception used as a debugging aid
    - the target is not the user of the program but the programmer
    - they **notify the programmer** that an error occurred but do not handle it,
    - users should never see assertion errors: assertions are automatically removed once the program is exported and no longer runs in debugging mode

Assertions
Example: Calculating Grades

- assume we are **given a class list** for a subject: each entry is a list of two parts
  - a list of first and last name for a student
  - a list of grades on assignments

```
test_grades = [[['peter', 'parker'], [10.0, 5.0, 85.0]],
               [['bruce', 'wayne'], [10.0, 8.0, 74.0]]]
```

- create a **new class list**, with names, grades, and an average

```
[[['peter', 'parker'], [10.0, 5.0, 85.0], 7.833333333333333],
 [['bruce', 'wayne'], [10.0, 8.0, 74.0], 8.466666666666667]]
```

Assertions Syntax

```
assert <condition>, <message>
```

Equivalent to:

```
if not <condition>:
    raise AssertionError(<message>)
```

```python
def avg(grades, name):
    assert len(grades) != 0, f"no grades data for {name}"
    return sum(grades) / len(grades)
```

raise AssertionError if list of grades is empty

program ends immediately if assertion is False

Where to Use Assertions?

- use **assertions** as ***debugging aid*** to
  - goal is to spot bugs as soon as introduced and make clear where they happened
  - check **types** of arguments or values (input to functions)
  - check that **invariants** on data structures are met
  - check **constraints** on return values (output of functions)
  - check for **violations** of constraints on procedure (e.g., no duplicates in a list)
- `assert False` is handy way to stop program

TAKE HOME MESSAGE

- a good programmer uses defensive programming to reduce the occurrence of bugs, and to make it easier to isolate and remove them
- **Exceptions** signal **invalid semantics**
  - Design your own **exception handlers** to recognize and react to bad user input
  - Prevent program execution from continuing
  - Unwinds call stack until reaches am exception handler
- **Assertions** flag when **internal properties** of program state aren't as expected
  - Implemented in Python as a special case of **exceptions**
  - Useful debugging aid, **should not be handled**

Back to Graphs

```python
class Digraph:
    """Represents a weighted directed graph using Nodes as keys."""

    def __init__(self, nodes=()):
        self._edges = {}  # dict: Node -> dict(Node -> weight)
        for node in nodes:
            self.add_node(node)

    def add_node(self, node):
        """Add a Node object to the graph."""
        if not isinstance(node, Node):
            raise TypeError("add_node expects a Node instance")
        if node in self._edges:
            raise ValueError(f"Duplicate node: {node.name}")
        self._edges[node] = {}

    def get_node(self, id):
        """Internal helper: resolve a node name or return node directly."""
        if isinstance(id, str) or isinstance(id, Node):
            for n in self._edges:
                if n.name == id:
                    return n
            raise ValueError(f"Unknown node name: '{id}'")
        raise TypeError(f"Expected Node or str, got {type(id).__name__}")

    def add_edge(self, src, dest, weight=1):
        """Add a directed edge between two Node objects (or names)."""
        src = self.get_node(src)
        dest = self.get_node(dest)
        self._edges[src][dest] = weight

    def get_all_nodes(self):
        return list(self._edges.keys())

    def outgoing_edges_of(self, node):
        return self._edges[node].copy()

    def children_of(self, node):
        return list(self._edges[node].keys())
```

```python
class Graph(Digraph):
    def add_edge(self, node1, node2, weight=1):
        super().add_edge(node1, node2, weight)
        super().add_edge(node2, node1, weight)
```

Specialized Graphs

```python
class MapGraph(Graph):
    """Graph that stores MapNode instances and offers geographic utilities."""

    EARTH_RADIUS_M = 6_371_000  # mean Earth radius in meters

    def __init__(self, nodes=(), max_speed_kmh=100):
        super().__init__(nodes)
        self.max_speed_kmh = max_speed_kmh

    def add_node(self, node):
        if not isinstance(node, MapNode):
            raise TypeError("MapGraph expects MapNode instances")
        super().add_node(node)

    def _resolve_mapnode(self, node):
        resolved = self.get_node(node)
        if not isinstance(resolved, MapNode):
            raise TypeError("MapGraph operations require MapNode instances but got: " + str(type(resolved)))
        return resolved

    def _coords(self, node):
        return self._resolve_mapnode(node).coords

    def haversine_distance(self, node1, node2):
        """Return the great-circle distance between two nodes in meters."""
        lon1, lat1 = self._coords(node1)
        lon2, lat2 = self._coords(node2)
        phi1, phi2 = radians(lat1), radians(lat2)
        dphi = radians(lat2 - lat1)
        dlambda = radians(lon2 - lon1)
        a = sin(dphi / 2)**2 + cos(phi1) * cos(phi2) * sin(dlambda / 2)**2
        c = 2 * atan2(sqrt(a), sqrt(1 - a))
        return MapGraph.EARTH_RADIUS_M * c

    def distance(self, node1, node2):
        """Return the Euclidean distance between two nodes in coordinate space."""
        x1, y1 = self._coords(node1)
        x2, y2 = self._coords(node2)
        return sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)

    def dist_great_circle(self, node1, node2):
        """Return the great-circle distance between nodes in kilometers."""
        return self.haversine_distance(node1, node2) / 1000

    def add_edge(self, node1, node2, weight=None):
        """
        Add an undirected edge whose weight equals the great-circle distance
        between the two endpoints (in kilometers).
        """
        node1_resolved = self._resolve_mapnode(node1)
        node2_resolved = self._resolve_mapnode(node2)
        actual_weight = self.distance(node1_resolved, node2_resolved)
        super().add_edge(node1_resolved, node2_resolved, actual_weight)
```

Building a Graph

```python
def build_city_nodes_graph():
    city_coords = {
        "Boston": (-71.0589, 42.3601),
        "Providence": (-71.4128, 41.8240),
        "New York": (-74.0060, 40.7128),
        "Chicago": (-87.6298, 41.8781),
        "Denver": (-104.9903, 39.7392),
        "Pittsburgh": (-79.9959, 40.4406),
        "Salt Lake City": (-111.8910, 40.7608),
        "San Francisco": (-122.4194, 37.7749),
        "Houston": (-95.3698, 29.7604),
        "Bozeman": (-111.0429, 45.6770),
        "Seattle": (-122.3321, 47.6062),
        "Minneapolis": (-93.2650, 44.9778),
        "Los Angeles": (-118.2437, 34.0522),
        "Atlanta": (-84.3880, 33.7490),
        "Miami": (-80.1918, 25.7617),
        "Philadelphia": (-75.1652, 39.9526),
        "Phoenix": (-112.0740, 33.4484),
        "San Diego": (-117.1611, 32.7157),
        "Dallas": (-96.7970, 32.7767),
        "Portland": (-122.6587, 45.5122),
        "Las Vegas": (-115.1398, 36.1699),
        "Austin": (-97.7431, 30.2672),
        "Nashville": (-86.7816, 36.1627),
        "Indianapolis": (-86.1581, 39.7684),
        "Charlotte": (-80.8431, 35.2271),
        "Cleveland": (-81.6944, 41.4993)
    }

    # Create MapNode objects for each city
    nodes = {name: MapNode(name, coords) for name, coords in city_coords

    # Create the graph with these nodes (no edges yet)
    g = MapGraph(nodes.values())
```

```python
g.add_edge('Boston', 'Cleveland')
g.add_edge('Boston', 'Providence')
g.add_edge('Boston', 'New York')

g.add_edge('Cleveland', 'Minneapolis')
g.add_edge('Cleveland', 'Chicago')
g.add_edge('Cleveland', 'Pittsburgh')

g.add_edge('Providence', 'Boston')
g.add_edge('Providence', 'New York')

g.add_edge('New York', 'Cleveland')
g.add_edge('New York', 'Pittsburgh')
g.add_edge('New York', 'Philadelphia')

g.add_edge('Philadelphia', 'Indianapolis')
g.add_edge('Philadelphia', 'Charlotte')

g.add_edge('Pittsburgh', 'Indianapolis')

g.add_edge('Chicago', 'Minneapolis')
g.add_edge('Chicago', 'Denver')
g.add_edge('Chicago', 'Indianapolis')

g.add_edge('Charlotte', 'Indianapolis')
g.add_edge('Charlotte', 'Nashville')
g.add_edge('Charlotte', 'Atlanta')

g.add_edge('Indianapolis', 'Denver')
g.add_edge('Indianapolis', 'Nashville')

g.add_edge('Minneapolis', 'Bozeman')

g.add_edge('Atlanta', 'Dallas')
g.add_edge('Atlanta', 'Miami')
g.add_edge('Atlanta', 'Houston')

g.add_edge('Miami', 'Houston')
```

Recap: Dijkstra's algorithm

- On the queue, tag each discovered states with its projected frontier so far
  - Remember that we're actually storing paths on the queue, so we can return one that reaches a goal
- True BFS frontier lies at state/path with smallest cost
  - Pick such a state/path to expand/extend
  - **Guaranteed that that is a shortest path**
  - So never put that state back on the queue
- When extending a path to a neighbor already on the queue, update the state's path and cost if the cost is lower

Not LIFO or FIFO, but a priority queue

Effectively in the visited set

NOT considered visited

```python
def dijkstra(graph, start, goal):
    # store best cost and path for discovered nodes that are
    # on present or future frontier
    queue = [(0, [start])]
    # separately, store nodes that are on past frontiers
    finished = set()

    while len(queue) > 0:
        print(f"Current queue: {queue}")

        # get a path off the true frontier
        cost, path = remove_min(queue)
        current_node = path[-1]
        finished.add(current_node)
        print(f"  Finished {current_node!r} with cost {cost}. Finished queue : {finished}")

        # optimality guaranteed for current node, return if goal
        if current_node == goal:
            return (cost, path)

        # update paths to neighbors on priority queue
        for edge in neighbors(graph, current_node):
            (next_node, weight) = edge
            if next_node not in finished:
                print(f"Processing {current_node!r}-->{next_node!r} with weight {weight}")
                new_cost = cost + weight
                new_path = path + [next_node]
                update_node(queue, next_node, new_cost, new_path)

        print()

    return None
```

Only terminate when priority queue is empty or if the true frontier contains the target

Take the node with the smallest cost (true frontier)

Expand the queue, except if the new path was already part of the true frontier (i.e., we know the shortest path already to that node)

If the next node wasn't part of the true frontier yet, we add it to the queue or update the path with its new cost

Dijkstra’s algorithm with Classes (essentially the same)

```python
def dijkstra_heap(graph, start, goal, visualize = False, pause=0.5):
    start_node = graph.get_node(start)
    goal_node = graph.get_node(goal)

    # Initialize heap (priority queue) with (cost, path)
    queue = [(0, [start_node])]
    heapq.heapify(queue)  # ensures it's a valid heap
    visited = set()

    while queue:
        # Pop the smallest-cost path
        cost, path = heapq.heappop(queue)
        current_node = path[-1]

        # Skip if already processed
        if current_node in visited:
            continue
        visited.add(current_node)
        print(f"✅ Finished {current_node!s} with cost {cost}. Finished set: {{"
              f"{', '.join(str(node) for node in visited)}}}")

        # Stop when we reach the goal
        if current_node == goal_node:
            return cost, path

        # Expand neighbors
        for neighbor, weight in graph.outgoing_edges_of(current_node).items():
            if neighbor in visited:
                continue
            new_cost = cost + weight
            new_path = path + [neighbor]
            heapq.heappush(queue, (new_cost, new_path))
            # print(f"  ➕ Added path {current_node!s} → {neighbor!s} (total cost {new_cost})")

        # Optional: pretty-print queue contents for debugging
        pretty_queue = [(c, [str(n) for n in p]) for c, p in queue]
        # print(f"Current queue: {pretty_queue}\n")

    # If no path found
    return None
```

A* search
(pronounced eigh/ay/ae/(h)ey-star)

Reducing wasteful exploration

- Dijkstra's is based on BFS principles
  - When expanding to neighbors, it has no idea whether it's getting closer or farther from goal
  - It only knows it's going away from the start as slowly as possible, to guarantee optimality
- Idea: augment cost for each state on the queue with an additional **estimate of cost-to-go**
  - Biases search towards expanding states ***that you think*** lie on an optimal path

The A* Search

- **Difficulty**: we want to still be able to generate the path with minimum cost

- A* is an algorithm that:
  - Uses heuristic to guide search
  - While ensuring that it will compute a path with minimum cost

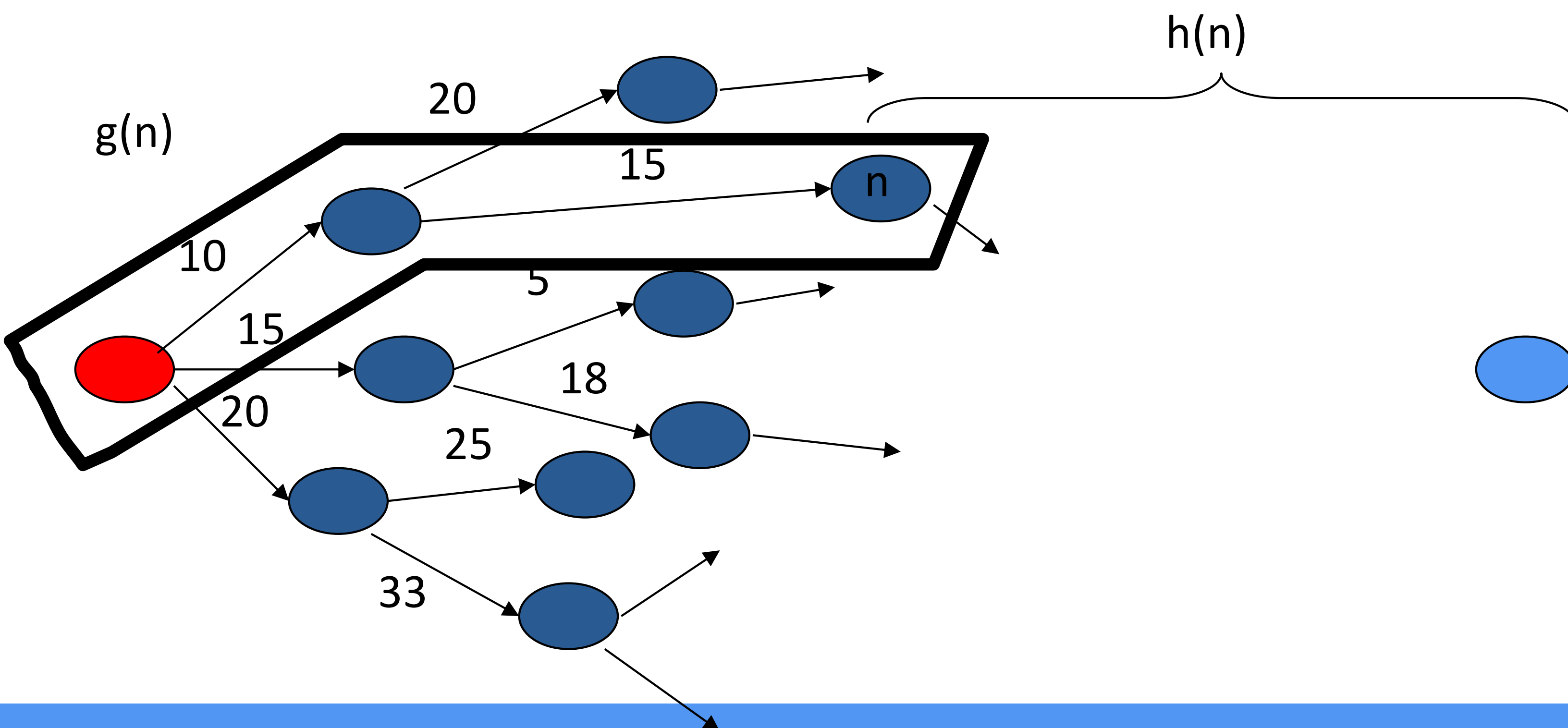
“estimated cost”

A* computes the function $f(n) = g(n) + h(n)$

“actual cost”

A*

- f(n) = g(n) + h(n)
  - g(n) = “cost from the starting node to reach n”
  - h(n) = “estimate of the cost of the cheapest path from n to the goal node”

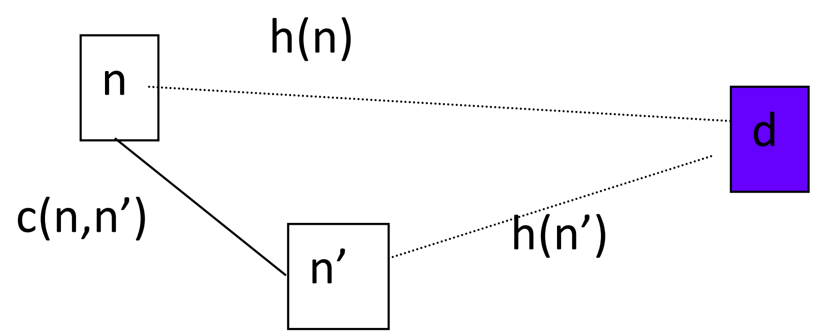


Properties of A*

- A* generates an optimal solution if h(n) is an admissible heuristic and the search space is a tree:
  - h(n) is **admissible** if it never overestimates the cost to reach the destination node

- A* generates an optimal solution if h(n) is a consistent heuristic and the search space is a graph:
  - h(n) is **consistent** if for every node n and for every successor node n' of n:

$$h(n) \leq c(n,n') + h(n')$$



- If h(n) is consistent then h(n) is admissible
- Frequently when h(n) is admissible, it is also consistent

Admissible Heuristics

- A heuristic is admissible if it is too optimistic, estimating the cost to be smaller than it actually is.

- Example:

  In the road map domain,

  h(n) = “Euclidean distance to destination”

  is admissible as normally cities are not connected by roads that make straight lines