
JEOPARDY!
The 6.100 Tour

Rules

Reset

- Reverse Jeopardy: The answers are the answers
- You have to answer as a group
- And go top to down per category
- Correct answers give candy points, wrong answers subtract candy points. And yes, you might owe me candy at the end ;)
- Every question has a certain number of seconds (indicated at the lower bar). After X seconds a bell indicates that the time is over
- Groups willing to answer the question must raise their hand
- Among groups with hands raised the moderator picks a group.
- At the discretion of the moderator, the answer is deemed correct or wrong. If correct, the group can choose the next question. If wrong, candy is subtracted and another group can answer

Classes, Methods, & Exceptions	A* (is out of scope for the mid-term)	More on Graphs
Yellow boxes were covered in class	4 Candy	4 Candy
3 Candy	5 Candy	5 Candy
4 Candy	8 Candy	6 Candy
5 Candy	10 Candy	12 Candy
6 Candy	12 Candy	4 Candy

2 Candy

Back

You are designing a simple system to model a Library with Books and Members.
You want:

- Each Library to keep track of its books and members.
- A Member to be able to borrow a Book.
- The Book itself should not know which library it belongs to (to keep it reusable).

Which of the following designs follows good object oriented design principles?

Option A

```
class Book:
    def __init__(self, title):
        self.title = title
        self.library = None

class Member:
    def __init__(self, name):
        self.name = name
        self.borrowed_books = []

class Library:
    def __init__(self):
        self.books = []
        self.members = []

    def add_book(self, book):
        book.library = self
        self.books.append(book)
```

Option B

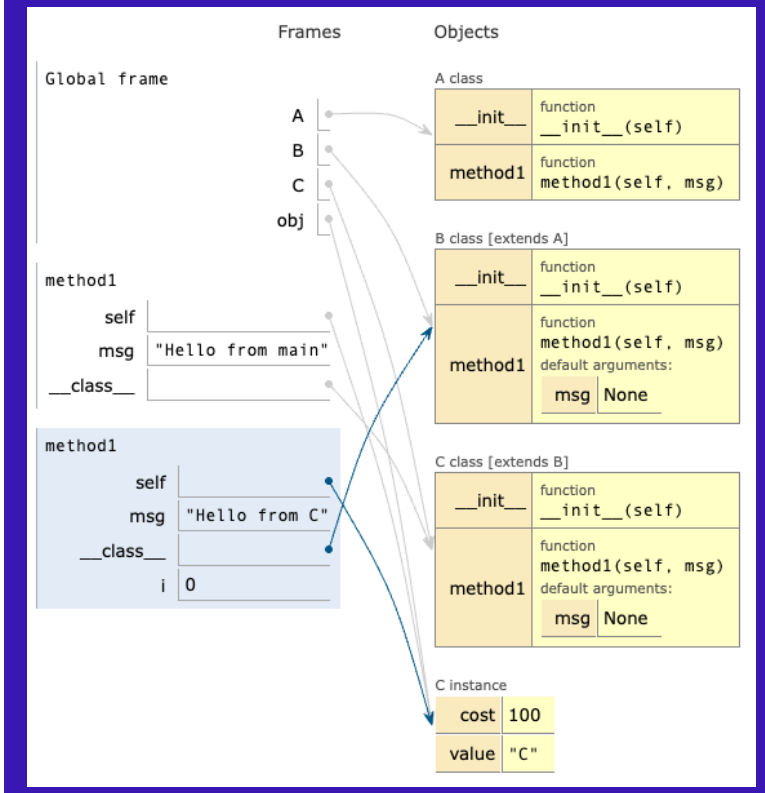
```
class Book:
    def __init__(self, title):
        self.title = title

class Member:
    def __init__(self, name):
        self.name = name
        self.borrowed_books = []

class Library:
    def __init__(self):
        self.books = []
        self.members = []

    def add_book(self, book):
        self.books.append(book)

    def register_member(self, member):
        self.members.append(member)

    def lend_book(self, member, book):
        if book in self.books and member in self.members:
            self.books.remove(book)
            member.borrowed_books.append(book)
```

Option C

```
class LibraryItem:
    def __init__(self, title):
        self.title = title
        self.borrowed_by = None

class Book(LibraryItem):
    def __init__(self, title, pages):
        super().__init__(title)
        self.pages = pages

class Member(Book):
    def __init__(self, title, pages, member_id):
        super().__init__(title, pages)
        self.member_id = member_id
```

3 Candy

Back



Option A

```
42  class A:
43      def __init__(self):
44          self.value = 'A'
45
46      def method1(self, msg):
47          print(msg)
48
49
50  class B(A):
51      def __init__(self):
52          super().__init__()
53          self.value = 'B'
54
55      def method1(self, msg=None):
56          super().method1("Hello from B")
57
58
59  class C(B):
60      def __init__(self):
61          super().__init__()
62          self.value = 'C'
63
64      def method1(self, msg=None):
65          super().method1("Hello from C")
66
67
68  obj = C()
69  obj.method1("Hello from main")
```

Option B

```
42  class A:
43      def __init__(self):
44          self.value = 'A'
45
46      def method1(self, msg):
47          print(msg)
48
49
50  class B(A):
51      def __init__(self):
52          super().__init__()
53          self.value = 'B'
54
55      def method1(self, msg=None):
56          for i in range(3):
57              super().method1("Hello from B")
58
59
60  class C(B):
61      def __init__(self):
62          super().__init__()
63          self.value = 'C'
64          self.cost = 100
65
66      def method1(self, msg=None):
67          super().method1("Hello from C")
68
69
70  obj = C()
71  obj.method1("Hello from main")
```

What code belongs to the frame on the right?
At what point in the execution was the snapshot of the frame taken?

4 Candy

Back

```python
def prepare_filename(filename):
    try:
        new_filename = filename.strip().lower()
        return new_filename
    except AttributeError:
        raise AttributeError("Invalid filename format")
    else:
        print("Filename prepared")

def read_and_parse(filename):
    file = None
    try:
        file = open(prepare_filename(filename), "r")
        text = file.read()
        if not text.strip():
            raise DataError("File is empty")
        number = int(text.strip())
        return number
    except ValueError:
        print("Inner except: Invalid number format.")
        raise DataError("Parsing failed")
    finally:
        print("Inner finally: Closing file.")
        if file:
            file.close()
```

```python
def process_data():
    try:
        number = read_and_parse("missing.txt")
        print("Processed number:", number + 10)
        return number
    except DataError as e:
        print("Outer except (DataError):", e)
    except Exception as e:
        print("Some error happened")
    finally:
        print("Cleanup complete.")

def app():
    try:
        result = process_data()
        print("App result:", result)
    except Exception as e:
        print("App error")
    else:
        print("All went fine!")

app()
```

What is the print output of calling app() assuming the file missing.txt does not exist?

Option A:
Filename prepared
Inner finally: Closing file.
Some error happened
Cleanup complete.
App result: None
All went fine!

Option B:
Inner finally: Closing file.
Some error happened
Cleanup complete.
App result: None
All went fine!

Option C:
Some error happened
Cleanup complete.
App result: None
All went fine!

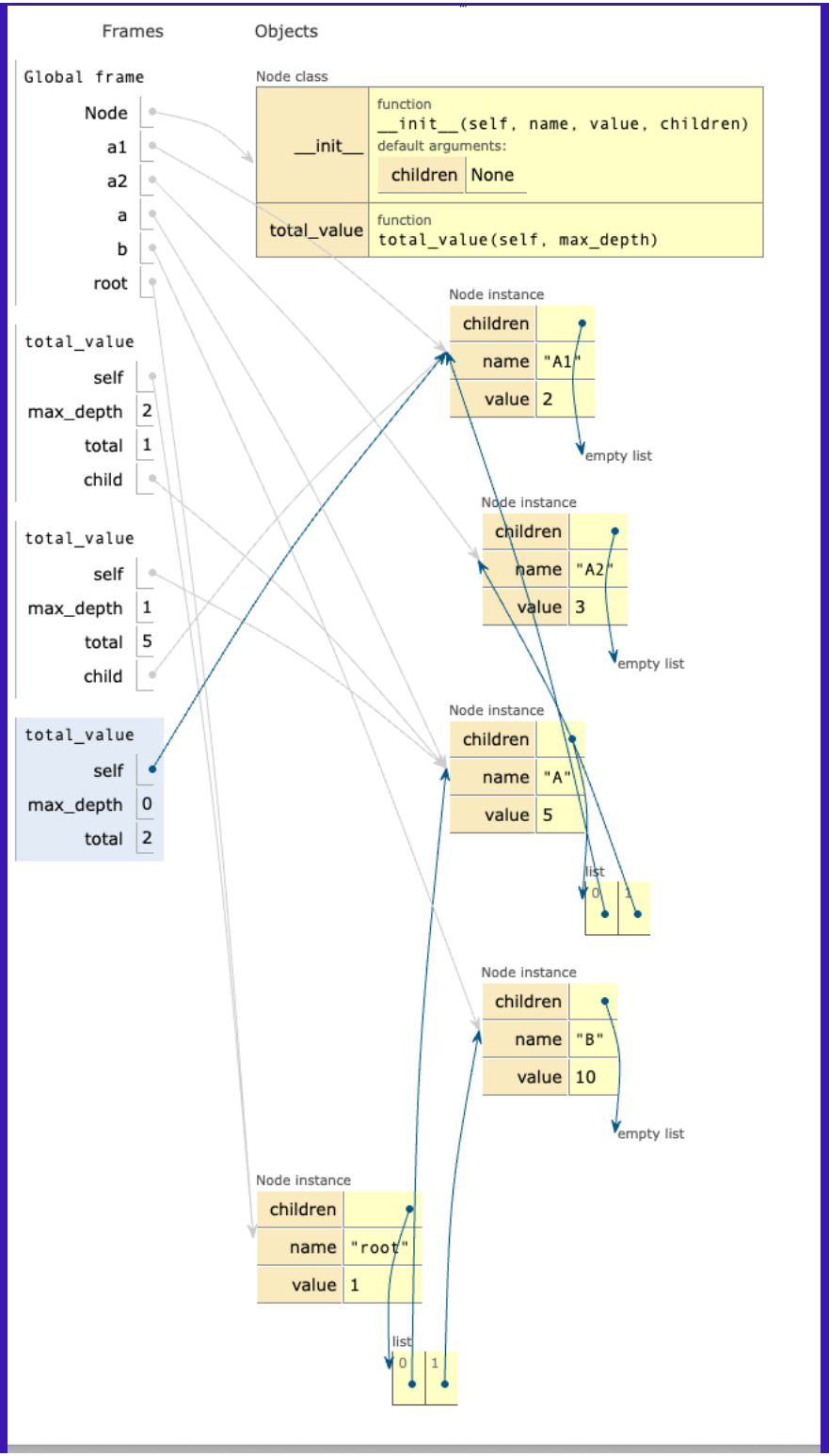
Option D:
Inner except: Invalid number format
Some error happened
Cleanup complete.
App result: None
All went fine!

5 Candy

Back

```python
class Node:
    def __init__(self, name, value, children=None):
        self.name = name
        self.value = value
        self.children = children if children else []

    def total_value(self, max_depth):
        total = self.value
        if max_depth > 0:
            for child in self.children:
                total += child.total_value(max_depth -1)
        else:
            print("Max depth reached - skipping anything below")
        return total

a1 = Node("A1", 2)
a2 = Node("A2", 3)
a = Node("A", 5, [a1, a2])
b = Node("B", 10)
root = Node("root", 1, [a, b])

print(root.total_value(2))
```

When the program prints “Max depth reached – skipping anything below” for the first time, what Python **frames** exists and what **objects** (you don’t need to show all the linkage between them)


Frames
Objects
Global frame
Node
a1
a2
a
b
root
total_value
self
max_depth 2
total 1
child
total_value
self
max_depth 1
total 5
child
total_value
self
max_depth 0
total 2
Node class
__init__
function
__init__(self, name, value, children)
default arguments:
children None
total_value
function
total_value(self, max_depth)
Node instance
children
name "A1"
value 2
empty list
Node instance
children
name "A2"
value 3
empty list
Node instance
children
name "A"
value 5
list
0 1
Node instance
children
name "B"
value 10
empty list
Node instance
children
name "root"
value 1
list
0 1

Back

6 Candy Bonus

```python
def average_per_subject(records):
    subject_totals = {}
    subject_counts = {}

    for record in records:
        _, grades, subjects = record
        for subject, grade in zip(subjects, grades):
            subject_totals[subject] = subject_totals.get(subject, 0) + grade
            subject_counts[subject] = subject_counts.get(subject, 0) + 1

    return {
        subject: subject_totals[subject] / subject_counts[subject]
        for subject in subject_totals
    }


def average_performance(records):
    total_sum = 0
    total_count = 0

    for record in records:
        _, grades, subjects = record
        if len(subjects) >0:
            total_sum += sum(grades) / len(grades)
            total_count += 1

    return total_sum / total_count if total_count else 0.0


grades = [
    [['peter', 'parker'], [10.0, 5.0, 8.5], ['math', 'science', 'english']],
    [['bruce', 'wayne'], [10.0, 8.0, 7.4], ['math', 'science', 'english']],
    [['pan', 'peter'], [], ['math', 'science', 'english']]
]

print("\nAverage per subject:")
for subject, avg in average_per_subject(grades).items():
    print(f"  {subject}: {avg:.2f}")

print(f"\nOverall total average: {average_performance(grades):.2f}")
```

Your program assumes that a student has a grade for every class. However, currently the developer doesn't use your method correctly and invokes it for classes, which don't have a grade yet.

How would you modify your code to make it safer?

Back

```python
def average_per_subject(records):
    subject_totals = {}
    subject_counts = {}
    for record in records:
        _, grades, subjects = record
        assert len(grades) == len(subjects), "They should be the same length"
        assert len(grades) > 0, "They shoudl be at least one subject with a grade"
        for subject, grade in zip(subjects, grades):
            subject_totals[subject] = subject_totals.get(subject, 0) + grade
            subject_counts[subject] = subject_counts.get(subject, 0) + 1

    return {
        subject: subject_totals[subject] / subject_counts[subject]
        for subject in subject_totals
    }


def average_performance(records):
    total_sum = 0
    total_count = 0
    for record in records:
        _, grades, subjects = record
        assert len(grades) == len(subjects), "They should be the same length"
        assert len(grades) > 0, "They shoudl be at least one subject with a grade"
        if len(subjects) >0:
            total_sum += sum(grades) / len(grades)
            total_count += 1

    return total_sum / total_count if total_count else 0.0


grades = [
    [['peter', 'parker'], [10.0, 5.0, 8.5], ['math', 'science', 'english']],
    [['bruce', 'wayne'], [10.0, 8.0, 7.4], ['math', 'science', 'english']],
    [['pan', 'peter'], [], ['math', 'science', 'english']]
]

print("\nAverage per subject:")
for subject, avg in average_per_subject(grades).items():
    print(f"  {subject}: {avg:.2f}")

print(f"\nOverall total average: {average_performance(grades):.2f}")
```

4 Candy

Back

Consider the following directed, weighted graph:

Edge	Weight
A → B	2
A → C	4
B → C	1
B → D	7
C → D	3

You run **Dijkstra's algorithm** starting from node **A with goal D**. After visiting nodes **A**, **B**, and **C**, what are the current shortest path in our queue?

```python
def dijkstra_heap(graph, start, goal, visualize = False, pause=0.5):
    start_node = graph.get_node(start)
    goal_node = graph.get_node(goal)
    queue = [(0, [start_node])]
    heapq.heapify(queue)
    visited = set()

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

        if current_node in visited:
            continue
        visited.add(current_node)

        if current_node == goal_node:
            return cost, path

        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))

    return None
```

Options:

A. [(6, ['A', 'B', 'C', 'D']), (9, ['A', 'B', 'D'])]

B. [(4, ['A', 'C']), (6, ['A', 'B', 'C', 'D']), (9, ['A', 'B', 'D'])] ✓

C. [(4, ['A', 'C']), (9, ['A', 'B', 'D'])]

D. [(4, ['A', 'C']), (6, ['A', 'B', 'C', 'D'])]

5 Candy

Back

Consider the map above with the goal to find the shortest path between A and B (Euclidian distance). Which nodes would the Dijkstra algorithm visit before finding the shortest path?

(A) All nodes as the connection C to A will be processed last.

(B) After two nodes (C, and then B)

(C) After it processed the second frontier [(C, D, F), then (B, E, G)] so 6 nodes

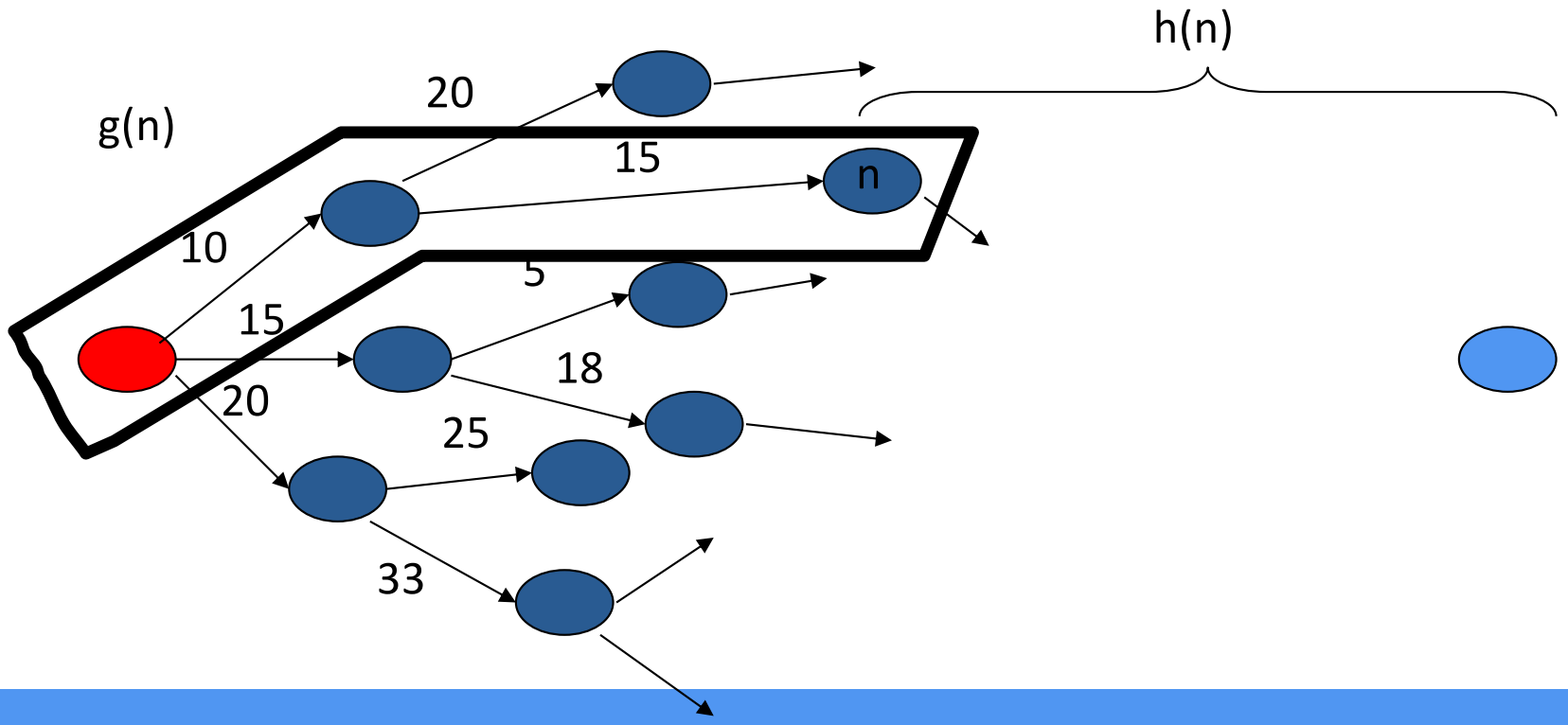
(D) It is a random algorithm: we don't know

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

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”



8 Candy

```python
def dijkstra_heap(graph, start, goal, visualize = False, pause=0.5):
    start_node = graph.get_node(start)
    goal_node = graph.get_node(goal)
    queue = [(0, [start_node])]
    heapq.heapify(queue)
    visited = set()

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

        if current_node in visited:
            continue
        visited.add(current_node)

        if current_node == goal_node:
            return cost, path

        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))

    return None
```

Modify the code to A* with h(n) being Euclidian distance to the goal

\

```python
def dijkstra_heap(graph, start, goal, visualize = False, pause=0.5):
    start_node = graph.get_node(start)
    goal_node = graph.get_node(goal)
    queue = [(0, [start_node])]
    heapq.heapify(queue)
    visited = set()

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

        if current_node in visited:
            continue
        visited.add(current_node)

        if current_node == goal_node:
            return cost, path

        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))

    return None
```

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

    g_score = 0
    h_score = graph.distance(start_node, goal_node)
    f_score = g_score + h_score
    queue = [(f_score, g_score, [start_node])]
    heapq.heapify(queue)
    visited = set()

    while queue:
        f_score, g_score, path = heapq.heappop(queue)
        current_node = path[-1]

        if current_node in visited:
            continue
        visited.add(current_node)

        if current_node == goal_node:
            return g_score, path

        for neighbor, edge_weight in graph.outgoing_edges_of(current_node).items():
            if neighbor in visited:
                continue
            new_g_score = g_score + edge_weight
            new_h_score = graph.distance(neighbor, goal_node)  # Heuristic estimate
            new_f_score = new_g_score + new_h_score
            new_path = path + [neighbor]
            heapq.heappush(queue, (new_f_score, new_g_score, new_path))

    return None
```

Back

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

    g_score = 0
    h_score = graph.distance(start_node, goal_node)
    f_score = g_score + h_score
    queue = [(f_score, g_score, [start_node])]
    heapq.heapify(queue)
    visited = set()

    while queue:
        f_score, g_score, path = heapq.heappop(queue)
        current_node = path[-1]

        if current_node in visited:
            continue
        visited.add(current_node)

        if current_node == goal_node:
            return g_score, path

        for neighbor, edge_weight in graph.outgoing_edges_of(current_node).items():
            if neighbor in visited:
                continue
            new_g_score = g_score + edge_weight
            new_h_score = graph.distance(neighbor, goal_node)  # Heuristic estimate
            new_f_score = new_g_score + new_h_score
            new_path = path + [neighbor]
            heapq.heappush(queue, (new_f_score, new_g_score, new_path))

    return None
```

12 Candy

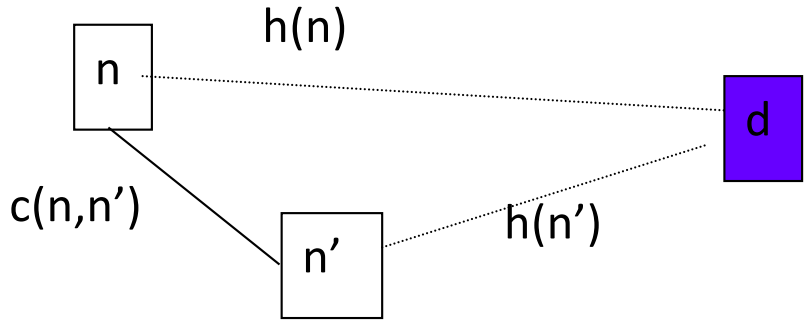
What are the requirements to the heuristic / cost estimate to guarantee that A* finds the optimal soluton?

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

Back

- 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

12 Candy

Back

Which of the following statements about the **A*** search algorithm is **true**?

A. A* always expands the fewest possible nodes among all optimal algorithms, regardless of the heuristic.

B. If the heuristic function $h(n)$is *admissible* but not *consistent*, A* may still find the optimal path but could re-expand nodes.

C. A* guarantees optimality only if the heuristic function overestimates the true cost to the goal.

4 Candy

Back

```python
def graph_algorithm(graph, start):
    start_node = graph.get_node(start)

    # Distance and predecessor tables
    distances = {node: float('inf') for node in graph.get_all_nodes()}
    distances[start_node] = 0
    predecessors = {node: None for node in graph.get_all_nodes()}

    queue = [(0, start_node)]
    heapq.heapify(queue)
    visited = set()

    while queue:
        cost, current_node = heapq.heappop(queue)
        if current_node in visited:
            continue
        visited.add(current_node)

        for neighbor, weight in graph.outgoing_edges_of(current_node).items():
            new_cost = cost + weight
            if new_cost < distances[neighbor]:
                distances[neighbor] = new_cost
                predecessors[neighbor] = current_node
                heapq.heappush(queue, (new_cost, neighbor))

    return distances, predecessors
```

What does this algorithm do?

What does the function return?

How does it compare to the Dijkstra algorithm from class?

Solution: Different implementation of Dijkstra using inf
Calculates all Values from source to any other node
Returns the predecessor of every node but needs to reconstruct the output

5 Candy

Back

```python
def dijkstra_heap_all(graph, start):
    start_node = graph.get_node(start)

    # Distance and predecessor tables
    distances = {node: float('inf') for node in graph.get_all_nodes()}
    distances[start_node] = 0
    predecessors = {node: None for node in graph.get_all_nodes()}

    queue = [(0, start_node)]
    heapq.heapify(queue)
    visited = set()

    while queue:
        cost, current_node = heapq.heappop(queue)
        if current_node in visited:
            continue
        visited.add(current_node)

        for neighbor, weight in graph.outgoing_edges_of(current_node).items():
            new_cost = cost + weight
            if new_cost < distances[neighbor]:
                distances[neighbor] = new_cost
                predecessors[neighbor] = current_node
                heapq.heappush(queue, (new_cost, neighbor))

    return distances, predecessors
```

To calculate the shortest path between every pair of nodes we could just invoke this function in a loop with every node as a source?

Would this be efficient?

How would you improve the algorithm?

Solution: You could cache the previous computed path

Floyd Warshall

Back

```python
def floyd_warshall(graph):
    """
    Compute shortest paths between all pairs of nodes using the Floyd-Warshall algorithm.
    Returns:
        distances[(u, v)] = shortest distance from u to v
        next_node[(u, v)] = next node on the shortest path from u to v
    """
    nodes = list(graph.get_all_nodes())
    distances = {}
    next_node = {}

    # Initialize distances with edge weights and 0 for self-loops
    for u in nodes:
        for v in nodes:
            if u == v:
                distances[(u, v)] = 0
            else:
                distances[(u, v)] = float('inf')
            next_node[(u, v)] = None

    # Fill in direct edge weights
    for u in nodes:
        for v, weight in graph.outgoing_edges_of(u).items():
            distances[(u, v)] = weight
            next_node[(u, v)] = v

    # Main triple loop
    for k in nodes:
        for i in nodes:
            for j in nodes:
                if distances[(i, k)] + distances[(k, j)] < distances[(i, j)]:
                    distances[(i, j)] = distances[(i, k)] + distances[(k, j)]
                    next_node[(i, j)] = next_node[(i, k)]

    return distances, next_node
```

Just an example, out of scope otherwise.

Good for dense networks

For directed graphs (doesn't take advantage of undirected graphs)

Can deal with negative weights

Back

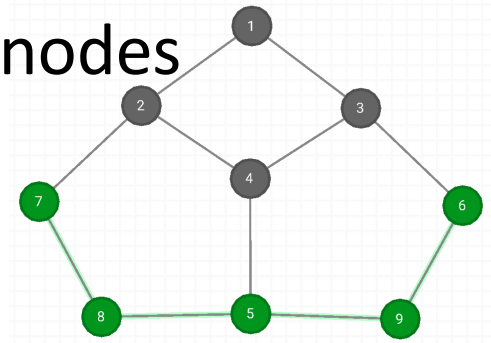
6 Candy

United Airlines wants to identify which routes between any two major airports require the greatest number of stops, indicating that those city pairs are not well served by existing connections?

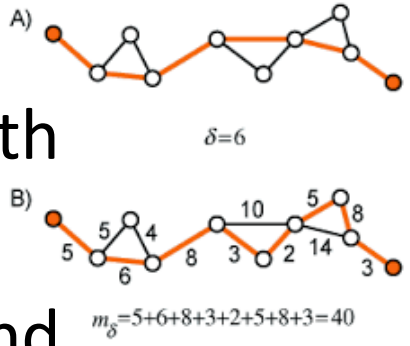
How would you calculate it?

All-pairs Shortest Path

- Compute shortest paths between all n^2 pairs of nodes (e.g., using Dijkstra or Floyd Warshall)
  - n is the number of nodes
  - Why is this useful?



- Consider the ***longest*** shortest path between any pair of nodes
  - Called the **diameter** of a graph



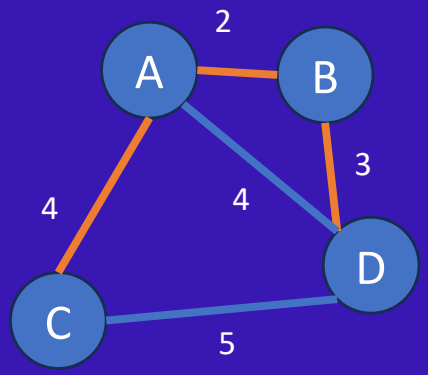
- In communication networks, this is the longest path any message has to travel, affects **latency**
- In the Facebook social graph, the diameter (in friend hops) is about **13**, but the **average shortest path length** is around **4–5** ("six degrees of separation").

Back

12 Candy

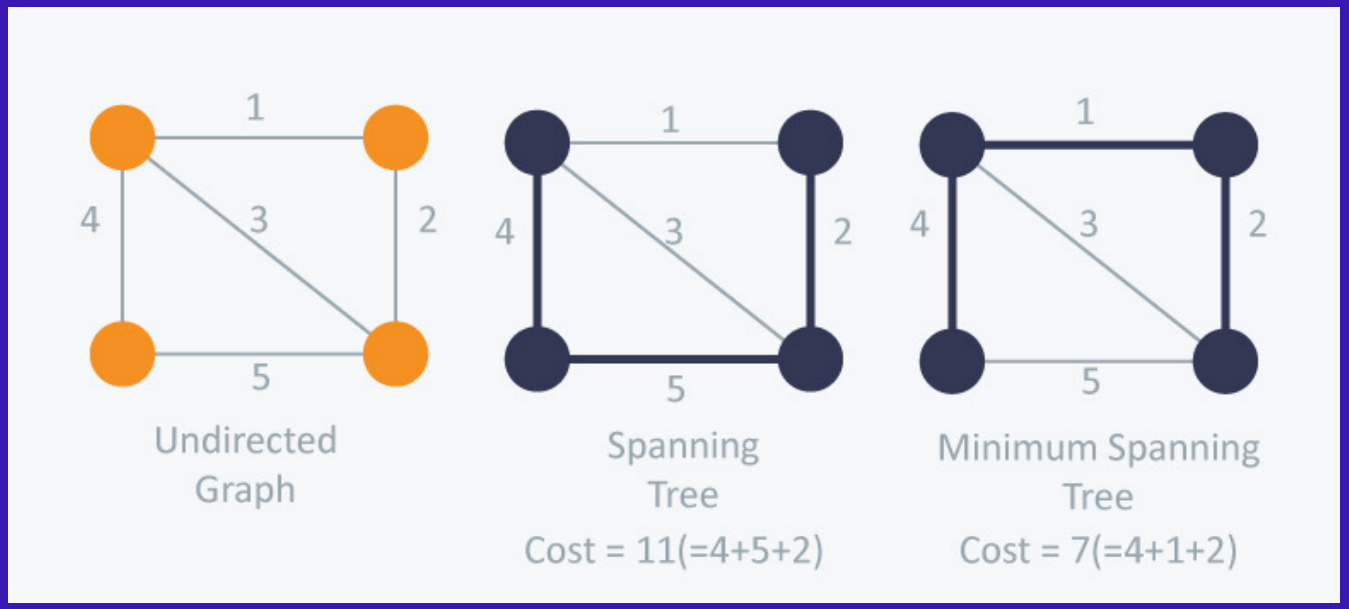
- **Imagine a city with lots of roads, but no bicycle lanes**
- **Assume that there is a non-negative cost associated with adding a lane to each road**
- **§ Objective function: Minimize the cost of adding bicycle lanes**

Design an algorithm to find the minimum number of bike-lanes so that it is possible to get between any pair of addresses using only bicycle lanes



Minimum Spanning Tree

Minimum Spanning Tree: Find a subset of edges of a connected graph that connects all nodes with the minimum total edge weight



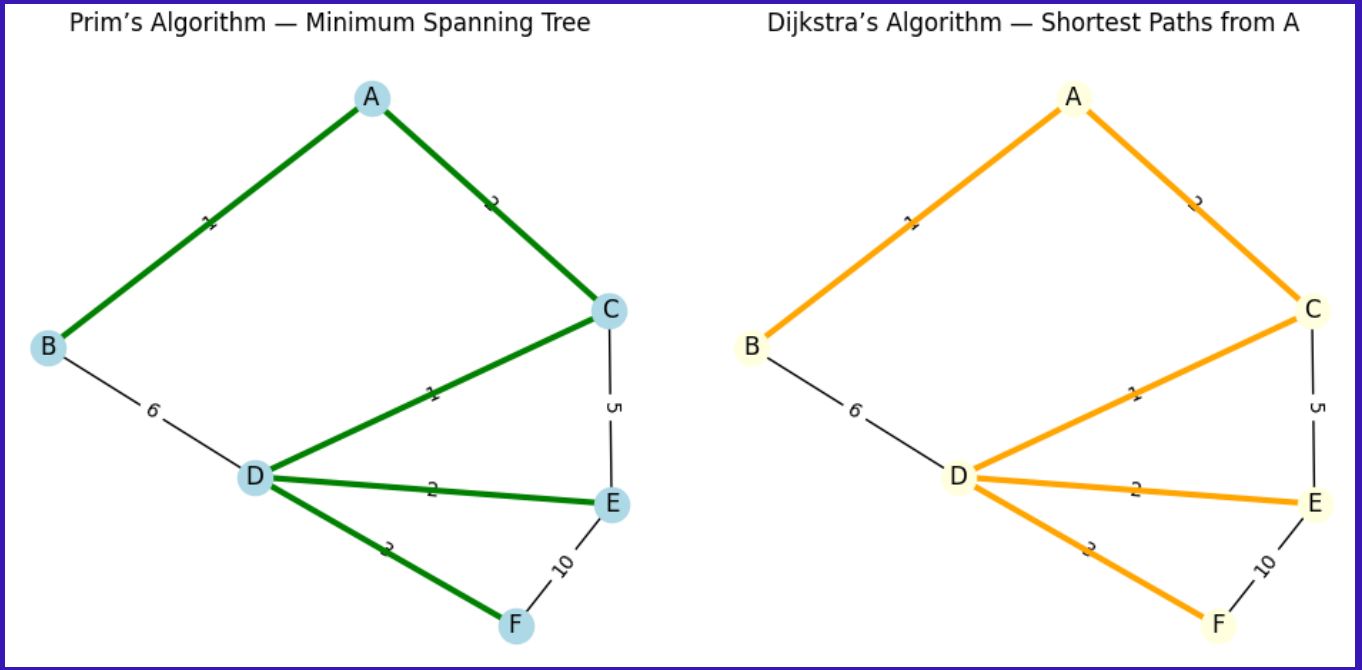
Prim Algorithm

```python
def prim(graph, start):
    start_node = graph.get_node(start)

    predecessors = {node: None for node in graph.get_all_nodes()}
    total_cost = 0

    queue = [(0, None, start_node)]
    heapq.heapify(queue)
    visited = set()

    while queue:
        cost, from_node, current_node = heapq.heappop(queue)
        if current_node in visited:
            continue
        visited.add(current_node)
        total_cost += cost

        if from_node is not None:
            predecessors[current_node] = from_node

        for neighbor, weight in graph.outgoing_edges_of(current_node).items():
            heapq.heappush(queue, (weight, current_node, neighbor))

    return total_cost, predecessors
```

Very similar to Dijkstra

4 Candy

Create an example where Dijkstra creates a different spanning tree than Prim's algorithm



Not a great example!

Back

Prim’s vs Dijkstra