Dijkstra Algorithm with two implementations:

Space complexity:
O(V)

Time Complexity:
O((V+E)logV)

```python
def find_shortest_path_with_heap(
        graph: dict[int, dict[int, float]],
        source: int,
        target: int
) -> tuple[list[int], float]:
    """...

    queue = HeapImp()
    queue.start(graph)
    queue.update(0, source)
    dist: dict[int, float] = {node:inf for node in graph}
    dist[source] = 0
    prev: dict[int, int] = {node:None for node in graph}

    while (queue):
        distance, node = queue.pop_min()

        if (node == target):
            break
        if (distance > dist[node]):
            continue

        for next,weight in graph[node].items():
            new_dist = dist[node] + weight
            if new_dist < dist[next]:
                dist[next] = new_dist
                prev[next] = node
                queue.update(new_dist, next)

    if (dist[target] == inf):
        return [], inf

    path: list[int] = []
    next: int = target
    while next is not None:
        path.append(next)
        next = prev[next]

    return path[::-1], dist[target]
```

queue: O(V)
dist: O(V)
prev: O(V)

O(1)
O(V)
O(logV)
O(V)

O(V) at worst
O(logV)

O(E): each edge

O(logV)

Reversing
O(V)

O(V) at worst

init:
O(1)

start:
O(n) Stores each node

push:
O(1)

update:
O(1)

swap:
O(1)

parent:
O(1)

children:
O(1)

pop-min:
O(1

```python
class HeapImp:
    def __init__(self):
        self.distances: list[float] = []
        self.nodes: list[int] = []
        self.indexes: dict[int, int] = {}

    def start(self, graph: dict[int, dict[int, float]]):
        for node in graph:
            self.distances.append(inf)
            self.nodes.append(node)
            self.indexes[node] = len(self.nodes) - 1
        return

    def push(self, distance: float, node: int):
        self.distances.append(distance)
        self.nodes.append(node)
        self.indexes[node] = len(self.nodes) - 1
        self.up(len(self.nodes) - 1)
        return

    def update(self, new_distance: float, node: int):
        index = self.indexes.get(node)
        old_distance = self.distances[index]
        self.distances[index] = new_distance

        if (new_distance < old_distance):
            self.up(index)
        else:
            self.down(index)
        return

    def swap(self, index1: int, index2: int):
        self.distances[index1], self.distances[index2] = self.distances[index2], self.distances[index1]
        self.nodes[index1], self.nodes[index2] = self.nodes[index2], self.nodes[index1]
        self.indexes[self.nodes[index1]] = index1
        self.indexes[self.nodes[index2]] = index2
        return
```

- init:
O(1)

- start:
O(n) : loops through each node in graph

- Push:
O(log n) inherited from up

- update:
O(log n) inherited from up or down

-swap : O(1)

```python
def parent(self, index: int) -> int:
    if (index == 0):
        return -1
    return (index - 1) // 2

def children(self, index: int) -> tuple[int, int]:
    return (2 * index + 1, 2 * index + 2)

def up(self, index: int):
    if (index == 0):
        return

    parent_index = self.parent(index)

    if self.distances[index] < self.distances[parent_index]:
        self.swap(index, parent_index)
        self.up(parent_index)

def pop_min(self) -> tuple[float, int]:
    if len(self.distances) == 0:
        raise IndexError("Trying to pop from an empty queue")

    min_distance = self.distances[0]
    min_node = self.nodes[0]
    self.distances[0] = self.distances[-1]
    self.nodes[0] = self.nodes[-1]
    self.indexes[self.nodes[0]] = 0

    self.distances.pop()
    self.nodes.pop()
    self.indexes.pop(min_node)

    self.down(0)

    return (min_distance, min_node)
```

-parent : O(1)

-children O(1)

-up :
At worst will loop log n of the list O(log n)

-pop_min :
O(log(n) inherited from down

Pown:
O(1)

```python
def down(self, index: int):
    size = len(self.distances)
    while True:
        left, right = self.children(index)
        smallest = index

        if left < size and self.distances[left] < self.distances[smallest]:
            smallest = left

        if right < size and self.distances[right] < self.distances[smallest]:
            smallest = right

        if smallest == index:
            break

        self.swap(index, smallest)
        index = smallest

    return
```

Pown: O(log n)
At worst will have to loop through half the items

Space complexity:
O(V)

```python
def find_shortest_path_with_array(
        graph: dict[int, dict[int, float]],
        source: int,
        target: int
) -> tuple[list[int], float]:
    """..."""
    queue = ArrayImp()
    queue.start(graph)
    queue.update(0, source)
    dist = {node:inf for node in graph}
    dist[source] = 0
    prev: dict[int, int] = {node:None for node in graph}

    while(queue):
        distance, node = queue.pop_min()

        if (node == target):
            break
        if (distance > dist[node]):
            continue

        for next,weight in graph[node].items():
            new_dist = dist[node] + weight
            if new_dist < dist[next]:
                dist[next] = new_dist
                prev[next] = node
                queue.update(new_dist, next)

    if (dist[target] == inf):
        return [], inf

    path: list[int] = []
    next: int = target
    while next is not None:
        path.append(next)
        next = prev[next]

    return path[::-1], dist[target]
```

Time Complexity:
O(V²+E)

queue: O(V)
dist: O(V)
prev: O(V)

O(1)
O(V)
O(1)
O(V)
O(V)

O(V) at worst
O(V)

O(E) each edge in the graph

O(1)

O(1)

O(N) at worst

reverse list
O(N)

```python
class ArrayImp:
    def __init__(self):
        self.distances: dict[int, float] = {}

    def start(self, graph:dict[int, dict[int, float]]):
        for node in graph:
            self.distances[node] = inf

    def pop_min(self) -> tuple[float,int]:
        min_node, min_distance = min(self.distances.items(), key=lambda item: item[1])
        del self.distances[min_node]
        return min_distance, min_node

    def update(self, new_distance: float, node: int):
        self.distances[node] = new_distance

    def push(self, new_distance: float, node: int):
        self.distances[node] = new_distance
```

init: O(1)

start: O(n) loop for each n

pop_min: O(n) loop through list at worst O(n)

update: O(1)

push: O(1)

N	density	# edges	“heap” time	“linear” time
1000	.01	10000	0.010689973831176758	0.08994483947753906
5000	.002	50000	0.041507720947265625	1.143932819366455
10000	.001	100000	0.0572962760925293	2.489440441131592
50000	.0002	500000	0.30506062507629395	56.342474937438965
100000	.0001	1000000	0.851081371307373	178.68678259849548

N	density	# edges	“heap” time	“linear” time
1000	1	999000	0.07222175598144531	0.09660625457763672
2000	1	3998000	0.3181722164154053	0.42501020431518555
3000	1	8997000	0.7484667301177979	0.993140459060669
4000	1	15996000	1.653285026550293	2.101970911026001
5000	1	24995000	2.689485788345337	3.4811010360717773
6000	1	35994000	4.042767763137817	5.135711193084717

As seen in the heap theoretical analysis and based off the data above, it outperforms the array implementation. This is because of the pop_min function, in the heap compared to the array. It has a runtime of O(log v) where the array is O(v). This change makes the heap implementation effectively better for larger vertices and lower densities data sets. However, with smaller data sets you can see that when the density is held to 1, the number of edges has little effect on runtime between the implementations. This is because the edges have less of an effect on the entire runtime across both implementations.