MEF University

Department of Computer Engineering

COMP 303 Analysis of Algorithm

Project 2

Comparison of Two Shortest Path Algorithms

Instructor: Prof.Dr. Adem Karahoca

Student: Kerem Ataç

Part 1: Shortest Path (Dijkstra's Algorithm)

1. Dijkstra's Algorithm

a - Describing of Dijkstra's Algorithm:

Dijkstra's algorithm is a **greedy algorithm** designed to find the shortest path from a starting source node to all other nodes (or a specific destination) **in a weighted graph with non-negative edge weights**. It works by maintaining a set of tentative distances to every node, **initialized to zero for the source and infinity for all others**, and **uses a priority queue (typically a min-heap) to iteratively select the unvisited node with the smallest known distance**. Once the closest node is selected, the algorithm examines its neighbors and performs a "relaxation" step: if the path to a neighbor through the current node is shorter than the previously recorded distance, the neighbor's distance is updated to this new lower value. This process repeats until the destination node is finalized or all reachable nodes have been visited, **guaranteeing the optimal path is determined**.

b - Determine Running time of the algorithm (Theoretical):

To determine the running time, we assign a cost (c) to each line and count how many times it executes based on the number of nodes (N) and edges (E).

Line	Pseudocode Step	Cost (ci)	Repetitions (approx.)	Total Cost
1	Initialize dist[S]=0, others =∞	c_1	N	$c_1 \cdot N$
2	Initialize Min-Heap Q with all nodes	c_2	1 (Build: O(N))	$c_2 \cdot N$
3	**While** Q is not empty:	c_3	N	$c_3 \cdot N$
4	u= Extract-Min(Q)	c_4	N (Cost is logN)	$c_4 \cdot N\log N$
5	For each neighbor v of u:	c_5	Total E times	$c_5 \cdot E$
6	alt=dist[u]+weight(u,v)	c_6	E	$c_6 \cdot E$

7	If alt<dist[v]:	c_7	E	$c_7 \cdot E$
8	dist[v]=alt	c_8	At most E	$c_8 \cdot E$
9	Decrease-Key(Q,v,alt)	c_9	At most E (Cost is logN)	$c_9 \cdot$ ElogN

Summing the total costs gives the function T(N):

$T(N)=(c_1+c_2+c_3)N+c_4 N\log N+(c_5+c_6+c_7+c_8)E+c_9 E\log N$

In Big-O notation, we **drop constants and lower-order terms**. The dominant terms are the heap operations (Extract-Min and Decrease-Key):

T(N) = O(NlogN+ElogN) = <u>O((N+E)logN)</u>

c - Steps of the algorithm for N=10, S=1, D=8:

Initialization
Source (S): Node 1, dist[1] = 0
Destination (D): Node 8
Unvisited set Q = {1,2,3,4,5,6,7,8,9,10}
Initial distances: dist[1] = 0, all others = ∞

Iteration 1
Select minimum node: 1 (dist = 0)
Neighbors: 2, 3, 4

Relaxation:
Node 2: 0 + w(1,2) = 0 + (1+2) = 3 → <u>dist[2] = 3, parent = 1</u>
Node 3: 0 + w(1,3) = 0 + (1+3) = 4 → dist[3] = 4, parent = 1
Node 4: 0 + w(1,4) = 0 + (1+4) = 5 → dist[4] = 5, parent = 1

Q: {2:3, 3:4, 4:5, others: ∞}

Iteration 2
Select minimum node: 2 (dist = 3)
Neighbors: 3, 4, 5

Relaxation:
Node 3: 3 + 5 = 8 > 4 → no update
Node 4: 3 + 6 = 9 > 5 → no update
Node 5: 3 + 7 = 10 → <u>dist[5] = 10, parent = 2</u>

Q: {3:4, 4:5, 5:10, others: ∞}

Iteration 3
Select minimum node: 3 (dist = 4)
Neighbors: 4, 5, 6

Relaxation:
Node 4: 4 + 7 = 11 > 5 → no update
Node 5: 4 + 8 = 12 > 10 → no update
Node 6: 4 + 9 = 13 → dist[6] = 13, parent = 3

Q: {4:5, 5:10, 6:13, others: ∞}

Iteration 4
Select minimum node: 4 (dist = 5)
Neighbors: 5, 6, 7

Relaxation:
Node 5: 5 + 9 = 14 > 10 → no update
Node 6: 5 + 10 = 15 > 13 → no update
Node 7: 5 + 11 = 16 → dist[7] = 16, parent = 4

Q: {5:10, 6:13, 7:16, others: ∞}

Iteration 5
Select minimum node: 5 (dist = 10)
Neighbors: 6, 7, 8

Relaxation:
Node 6: 10 + 11 = 21 > 13 → no update
Node 7: 10 + 12 = 22 > 16 → no update
Node 8: 10 + 13 = 23 → <u>dist[8] = 23, parent = 5</u>

Q: {6:13, 7:16, 8:23, others: ∞}

Iteration 6
Select minimum node: 6 (dist = 13)
Neighbors: 7, 8, 9

Relaxation:
Node 7: 13 + 13 = 26 > 16 → no update
Node 8: 13 + 14 = 27 > 23 → no update
Node 9: 13 + 15 = 28 → dist[9] = 28, parent = 6

Q: {7:16, 8:23, 9:28}

Iteration 7
Select minimum node: 7 (dist = 16)
Neighbors: 8, 9, 10

Relaxation:
Node 8: 16 + 15 = 31 > 23 → no update
Node 9: 16 + 16 = 32 > 28 → no update
Node 10: 16 + 17 = 33 → dist[10] = 33, parent = 7

Q: {8:23, 9:28, 10:33}

Iteration 8
Select minimum node: 8 (dist = 23)
Destination reached. Stop.

Final Output
With backtracking the first **who updated the dist[8] = 23, and parent = 5**. Secondly, who **updated dist[5] =10, and parent = 2.** This goes to the beginning and we determined the shortest path. Underlined texts in the iterations show the backtracking.
Shortest path: 1 → 2 → 5 → 8
Total cost: 23

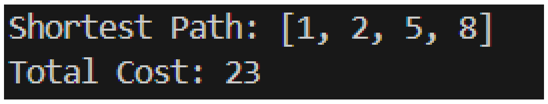
2. Implementing of Dijkstra's Algorithm

Code:

```
import heapq                                                # 1
import sys                                                  # 1

def dijkstra_basic(N, S, D):                                # 1
    """
    Finds the shortest path from Source S to Destination D.
    Returns the path and total cost.
    """
    # Initialize distances to infinity
    # Using a dictionary for sparse storage, though array is fine for 1..N
    dist = {i: float('inf') for i in range(1, N + 1)} # 1 (N internal
iterations)
    prev = {i: None for i in range(1, N + 1)}      # 1 (N internal
iterations)

    dist[S] = 0                                             # 1
```

```
# Priority Queue (Min-Heap): Stores tuples (distance, node)
# The heap is ordered by the first element (distance)
pq = [(0, S)]                                       # 1

while pq:                                           # Approx N
    # Extract the node with the smallest distance
    current_dist, u = heapq.heappop(pq)             # Approx N

    # If we reached the destination, we can stop
    if u == D:                                      # Approx N
        break                                       # 0 or 1

    # If the popped node is stale (we found a shorter path later),skip
    if current_dist > dist[u]:                      # Approx N
        continue                                    # 0 to Approx N

    # Determine neighbors: |i-j| <= 3
    # Range is bounded by 1 and N
    start_node = max(1, u - 3)                      # Approx N
    end_node = min(N, u + 3)                        # Approx N

    for v in range(start_node, end_node + 1):       # Max 7 per visited node
        if u == v:                                  # Max 7 per visited node
            continue                                # Max 1 per visited node

        # Weight formula: w_ij = i + j
        weight = u + v                              # Max 6 per visited node

        # Relaxation step
        if dist[u] + weight < dist[v]:              # Max 6 per visited node
            dist[v] = dist[u] + weight              # Max 6 per visited node
            prev[v] = u                             # Max 6 per visited node
            heapq.heappush(pq, (dist[v], v))        # Max 6 per visited node

# Reconstruct path
path = []                                           # 1
curr = D                                            # 1
if dist[D] == float('inf'):                         # 1
    return [], float('inf')                         # 0 or 1
    while curr is not None:                                 # Path Length (<= N)
        path.append(curr)                                   # Path Length (<= N)
        curr = prev[curr]                                   # Path Length (<= N)
    path.reverse()                                          # 1

    return path, dist[D]                                    # 1

# Example run for N=10, S=1, D=8
if __name__ == "__main__":                                  # 1
    n_val, s_val, d_val = 10, 1, 8                          # 1
    path, cost = dijkstra_basic(n_val, s_val, d_val)  # 1
    print(f"Shortest Path: {path}")                         # 1
    print(f"Total Cost: {cost}")                            # 1
```

Output:



```
Shortest Path: [1, 2, 5, 8]
Total Cost: 23
```

3. Measuring Running time

a - Adding counters in the code to measure the number of repetitions:

- Initialized a counter variable at the start of the function.

```
def dijkstra_search(N, S, D):
    # --- NEW: Initialize Counter ---
    total_repetitions = 0
```

- Incremented the counter inside the main while loop (processing a node).

```
    while pq:
        # --- NEW: Count Outer Loop (Node Processing) ---
        total_repetitions += 1
```

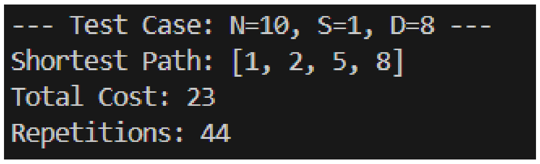
- Incremented the counter inside the inner for loop (checking an edge).

```
        for v in range(start_node, end_node + 1):
            if u == v: continue
            # --- NEW: Count Inner Loop (Edge Checking) ---
            total_repetitions += 1
```

- Returned the counter at the end so it can be analyzed

```
    # --- NEW: Return the counter along with path and cost ---
    return path, dist[D], total_repetitions
```

Output:



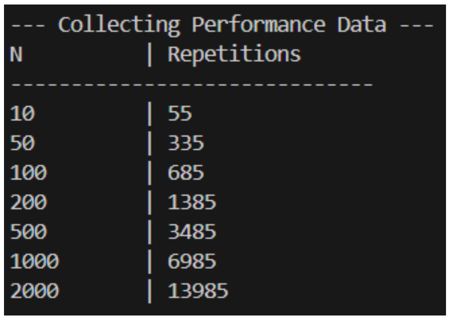
```
--- Test Case: N=10, S=1, D=8 ---
Shortest Path: [1, 2, 5, 8]
Total Cost: 23
Repetitions: 44
```

b - Running the program for N=(10, 50, 100, 200, 500, 1000, 2000) with S=1, D=N:

- Created a list of test values.
- Added a loop to run the dijkstra_search function for each N.
- Stored the results (total_repetitions) for later use.

```
if __name__ == "__main__":
    # Define the N values required by the project
    test_values = [10, 50, 100, 200, 500, 1000, 2000]

    # Store results for graphing later
    measured_data = []

    print(f"{'N':<10} | {'Repetitions':<15}")
    print("-" * 30)

    for N in test_values:
        # Run with S=1 and D=N as required
        path, cost, reps = dijkstra_search(N, 1, N)

        # Save the 'reps' (repetition count) for the graph
        measured_data.append(reps)

        print(f"{N:<10} | {reps:<15}")
```

Output:



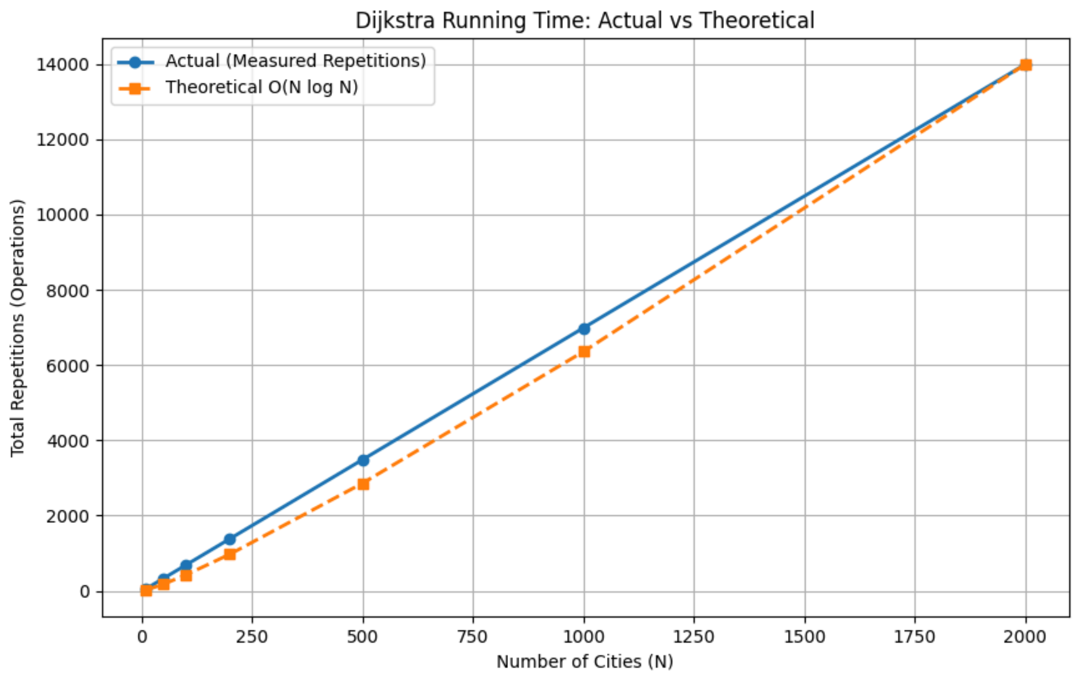
```
--- Collecting Performance Data ---
N          | Repetitions
------------------------------
10         | 55
50         | 335
100        | 685
200        | 1385
500        | 3485
1000       | 6985
2000       | 13985
```

c - Drawing and Comparing of measured and theoretical running times:

- Imported matplotlib for plotting.
- Calculated Theoretical Data was O((N+E)log N) from previous calculations. Since graph sparse is **(E≅6N)**, the theoretical complexity is roughly **O(NlogN)**.
- Plotted two lines: One for the **measured_data** and one for the **theoretical_data**.

```
import matplotlib.pyplot as plt
import math

# 1. Theoretical Calculation (O(N log N))
# We compute a scaling factor 'k' to align the curves visually for
comparison
# Using the largest N (2000) to find k: k = measured_2000 / (2000 *
log2(2000))
last_n = test_values[-1]
last_meas = measured_data[-1]
k = last_meas / (last_n * math.log2(last_n))

theoretical_data = [k * n * math.log2(n) for n in test_values]

# 2. Plotting
plt.figure(figsize=(10, 6))

# Plot Actual (Measured)
plt.plot(test_values, measured_data, marker='o', label='Actual (Measured
Repetitions)')

# Plot Theoretical
plt.plot(test_values, theoretical_data, marker='s', linestyle='--',
label='Theoretical O(N log N)')

plt.title('Dijkstra Running Time: Actual vs Theoretical')
plt.xlabel('Number of Cities (N)')
plt.ylabel('Total Repetitions')
plt.legend()
plt.grid(True)
plt.show()
```

Output:



d - Interpreting of results:

Linear Growth of Actual Repetitions:
The blue line ("Actual") is almost perfectly straight (linear). This happens because our counter tracks the number of loop iterations (how many nodes popped + how many neighbors checked).

- Since every node has at most 6 neighbors (|i-j| ≤ 3), the total number of edges (E) is roughly 6N.
- Therefore, the total repetitions are approximately N + 6N = 7N, which is a **linear function O(N)**.

Comparison with Theoretical Curve:
The orange dashed line represents the theoretical complexity O(N log N). It is slightly curved compared to the straight blue line.

- The theoretical curve grows faster than the measured line (in terms of slope change) because it accounts for the cost of Heap operations (heappush/heappop).
- Our code's counter treats every loop iteration as taking "1 unit" of time. In reality, as N grows, the heap operations inside those loops take slightly longer O(log N) to execute.
- Therefore, while the number of repetitions is linear (blue line), the actual CPU time (if measured in seconds) would likely follow the curve of the orange line more closely.

Part 2: A* Search Algorithm

1. A* Search Algorithm

a - Describing of A* Search Algorithm:

The A* (A-Star) algorithm is a **best-first search algorithm** that finds the shortest path from a starting source node to a destination node. It **improves upon Dijkstra's algorithm** by utilizing a **heuristic function**, h(n), which **estimates the cost from the current node n to the goal**.
The algorithm **selects nodes based on the cost function f(n) = g(n) + h(n)**
Where:

- g(n): The **actual cost from the start node to node n** (same as Dijkstra).
- h(n): The **estimated cost from node n to the destination**.

By prioritizing nodes with the lowest f(n), A* guides the **search towards the destination more intelligently** than Dijkstra, provided the heuristic is **admissible** (never overestimates the true cost).

b - Determine Running time of the algorithm (Theoretical):

Line	Pseudocode Step	Cost (ci)	Repetitions (approx.)	Total Cost
1	Initialize g[S]=0, others=∞, f[S]=h(S,D)	c_1	N	$c_1 \cdot N$
2	Initialize Min-Heap Q with start node (f[S],S)	c_2	1	c_2
3	While Q is not empty:	c_3	N	$c_3 \cdot N$
4	u= Extract-Min(Q)	c_4	N (Cost is logN)	$c_4 \cdot N\log N$
5	If u==D: Break	c_5	1	c_5
6	For each neighbor v of u:	c_6	Total E times	$c_6 \cdot E$
7	tentative_g=g[u]+weight(u,v)	c_7	E	$c_7 \cdot E$
8	If tentative_g<g[v]:	c_8	E	$c_8 \cdot E$
9	g[v]=tentative_g, f[v]=g[v]+h(v,D)	c_9	At most E	$c_9 \cdot E$
10	Insert/Decrease-Key(Q, v, f[v])	c_{10}	At most E (Cost is logN)	$c_{10} \cdot E\log N$

Summing the total costs gives the function T(N):

$T(N)=(c_1+c_3)N+c_4N\log N+(c_6+c_7+c_8+c_9)E+c_{10}E\log N$

In Big-O notation, we **drop constants and lower-order terms**. The dominant terms are the heap operations (Extract-Min and Decrease-Key):

$T(N) = O(N\log N+E\log N)$ = <u>$O((N+E)\log N)$</u>

The theoretical running time of A* **depends heavily on the quality of the heuristic**.

- **Worst Case: If the heuristic is not informative (h(n)=0)**, **A* behaves exactly like Dijkstra's algorithm**. The complexity is determined by the number of edges (E) and nodes (N) processed using the priority queue.
- **Time Complexity: O((N+E)log N)** or **roughly O(Nlog N)** for sparse graphs like the one in this project where **E≅6N**.

c - Steps of the algorithm for N=8, S=1, D=8:

h(i, j) = |j - i|

Initialization
Source (S): Node 1, g[1]=0, h[1]=|8-1|=7, f[1]=7
Destination (D): Node 8
Priority Queue Q = {(7, 1)}
All other g values = ∞

Iteration 1
Select minimum node: 1 (f=7, g=0)
Neighbors: 2, 3, 4
Relaxation:
Node 2: g=0+3=3, h=|8-2|=6, f=9. Parent=1.
Node 3: g=0+4=4, h=|8-3|=5, f=9. Parent=1.
Node 4: g=0+5=5, h=|8-4|=4, f=9. Parent=1.

Q: {2:9, 3:9, 4:9} (**Ties broken by index**)

Iteration 2
Select minimum node: 2 (f=9, g=3)
Neighbors: 3, 4, 5

Relaxation:
Node 3: 3+5=8 > 4 *(previous dist[3] = 4)* No update.
Node 4: 3+6=9 > 5 No update.
Node 5: 3+7=10, h=|8-5|=3, f=13. Parent=2.

Q: {3:9, 4:9, 5:13}

Iteration 3
Select minimum node: 3 (f=9, g=4)
Neighbors: 4, 5, 6

Relaxation:
Node 4: 4+7=11 > 5 No update.

Node 5: 4+8=12 > 10 No update.
Node 6: 4+9=13, h=|8-6|=2, f=15. Parent=3. Update.

Q: {4:9, 5:13, 6:15}

Iteration 4
Select minimum node: 4 (f=9, g=5)
Neighbors: 5, 6, 7

Relaxation:
Node 5: 5+9=14 > 10 No update.
Node 6: 5+10=15 > 13 No update.
Node 7: 5+11=16, h=|8-7|=1, f=17. Parent=4.

Q: {5:13, 6:15, 7:17}

Iteration 5
Select minimum node: 5 (f=13, g=10)
Neighbors: 6, 7, 8

Relaxation:
Node 6: 10+11=21 > 13 No update.
Node 7: 10+12=22 > 16 No update.
Node 8: 10+13=23, h=|8-8|=0, f=23. Parent=5.

Q: {6:15, 7:17, 8:23}

Iteration 6

Select minimum node: 6 (f=15, g=13)
Neighbors: 7, 8

Relaxation:
Node 7: 13+13=26 > 16 No update.
Node 8: 13+14=27 > 23 No update.

Q: {7:17, 8:23}

Iteration 7
Select minimum node: 7 (f=17, g=16)
Neighbors: 8

Relaxation:
Node 8: 16+15=31 > 23 No update.

Q: {8:23}

Iteration 8
Select minimum node: 8 (f=23, g=23)
Destination reached. Stop.

Final Output
Backtracking: 8 → 5 → 2 → 1
Shortest path: 1 → 2 → 5 → 8
Total Cost: 23

The **"Heuristic" (the guess of how far is left) is very weak compared to the "Weights"** (the cost to travel). **Because the Heuristic is so small**, it doesn't have enough "power" to drastically change the order of nodes compared to Dijkstra. **Therefore, A* ends up following almost the exact same path as Dijkstra**, making the steps look nearly identical.

Hypothetical Example:

If the edge to Node 4 cost 4 instead of 5:

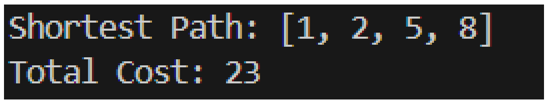
- **Dijkstra: Sees g=4. Still thinks Node 2 (g=3) is better. Picks Node 2.**
- ***A****: Sees g=4, h=4, so f=8. Sees Node 2 is f=9. Picks Node 4.**

2. Implementing of A* Search Algorithm

Code:

```
import heapq                                          # 1
import sys                                            # 1

def astar_basic(N, S, D):                             # 1
    """
    Finds the shortest path using A* Algorithm
    Returns the path and total cost.
    """

    # Heuristic function: h(i, j) = abs(i - j)
    def heuristic(u, v):
        return abs(u - v)                             # 1 (Call cost O(1))

    # Initialize g_scores (actual distance) to infinity
    g_score = {i: float('inf') for i in range(1, N + 1)} # 1 (N
iterations)

    # Store parents for path reconstruction
    prev = {i: None for i in range(1, N + 1)}            # 1 (N
iterations)

    g_score[S] = 0                            # 1

    # Priority Queue: Stores tuples (f_score, node)
    # f_score = g_score + h_score
    initial_h = heuristic(S, D)               # 1
    pq = [(initial_h, S)]                     # 1

    while pq:                                 # Approx N
        # Extract node with smallest f_score
        current_f, u = heapq.heappop(pq)      # Approx N (log N cost)

        if u == D:                            # 1
            break                             # 1

        # If we found a shorter path to u elsewhere, skip stale entry
        # Note: current_f is f-score, check against g_score[u] + h(u,D)
        # Simplified check: if popped f > known g + h
        if current_f > g_score[u] + heuristic(u, D): # 1
            continue

        # Determine neighbors: |i - j| <= 3
        start_node = max(1, u - 3)            # Approx N
        end_node = min(N, u + 3)              # Approx N

        for v in range(start_node, end_node + 1): # Max 7 per visited node
            if u == v:
                continue                      # Max 7 per visited node

            # Weight formula: w_ij = i + j
            weight = u + v                    # Max 6 per visited node

            # Tentative g_score
            tentative_g = g_score[u] + weight # Max 6 per visited node
            # Relaxation step
            if tentative_g < g_score[v]:      # Max 6 per visited node
                g_score[v] = tentative_g      # Max 6 per visited node
                prev[v] = u                   # Max 6 per visited node

                f_score = tentative_g + heuristic(v, D) # Max 6 per
visited node
                heapq.heappush(pq, (f_score, v))        # Max 6 per
visited node

    # Reconstruct path
    path = []
    curr = D                                  # 1

    if g_score[D] == float('inf'):            # 1
        return [], float('inf')               # 0 or 1

    while curr is not None:                   # Path Length (<= N)
        path.append(curr)                     # Path Length (<= N)
        curr = prev[curr]                     # Path Length (<= N)

    path.reverse()                            # 1
    return path, g_score[D]                   # 1

# Example run for N=8, S=1, D=8
if __name__ == "__main__":                    # 1
    n_val, s_val, d_val = 8, 1, 8             # 1
    path, cost = astar_basic(n_val, s_val, d_val)  # 1
    print(f"Shortest Path: {path}")           # 1
    print(f"Total Cost: {cost}")              # 1
```

Output:



```
Shortest Path: [1, 2, 5, 8]
Total Cost: 23
```

The current heuristic **h(i, j) = |i-j| is very weak** because the **edge weights (i+j) are significantly larger** than the distance in steps (1 step cost vs ≅ 2i).

A more aggressive admissible heuristic could be:
h_{alt}(u, D) = (D - u) x 2
Rationale: To travel from u to D, we need at least one edge. The minimum edge weight involving any node k is roughly 2k. Even conservatively, the smallest edge weight is always ≥ 3 (for nodes 1 and 2). Thus, scaling the distance by the minimum possible edge weight would guide the search faster while remaining admissible.

3. Measuring Running time

a - Adding counters in the code to measure the number of repetitions:

- Initialized a counter variable at the start of the function.

```
def astar_search(N, S, D):
    # NEW: Initialize Counter
    total_repetitions = 0
```

- Incremented the counter inside the main while loop (processing a node).

```
    while pq:
        # NEW: Count Outer Loop (Node Processing)
        total_repetitions += 1
```

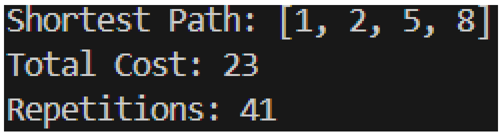
- Incremented the counter inside the inner for loop (checking an edge).

```
        for v in range(start_node, end_node + 1):
            if u == v: continue

            # NEW: Count Inner Loop (Edge Checking)
            total_repetitions += 1
```

- Returned the counter at the end so it can be analyzed

```
    # NEW: Return the counter along with path and cost
    return [], g_score[D], total_repetitions
```

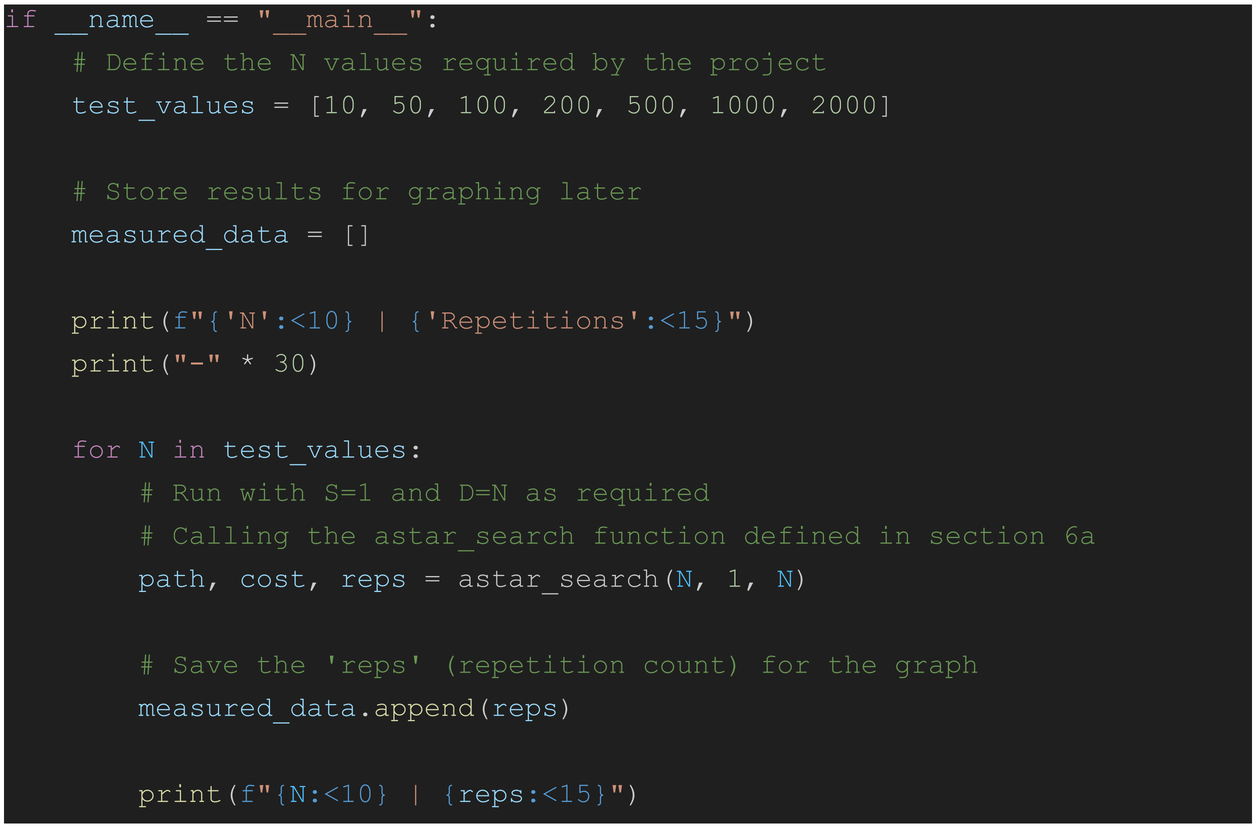
Output:



```
Shortest Path: [1, 2, 5, 8]
Total Cost: 23
Repetitions: 41
```

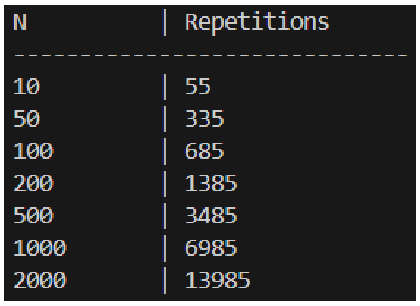
b - Running the program for N=(10, 50, 100, 200, 500, 1000, 2000) with S=1, D=N:

- Created a list of test values.
- Added a loop to run the astar_search function for each N.
- Stored the results (total_repetitions) for later use.



```python
if __name__ == "__main__":
    # Define the N values required by the project
    test_values = [10, 50, 100, 200, 500, 1000, 2000]

    # Store results for graphing later
    measured_data = []

    print(f"{'N':<10} | {'Repetitions':<15}")
    print("-" * 30)

    for N in test_values:
        # Run with S=1 and D=N as required
        # Calling the astar_search function defined in section 6a
        path, cost, reps = astar_search(N, 1, N)

        # Save the 'reps' (repetition count) for the graph
        measured_data.append(reps)

        print(f"{N:<10} | {reps:<15}")
```

Output:



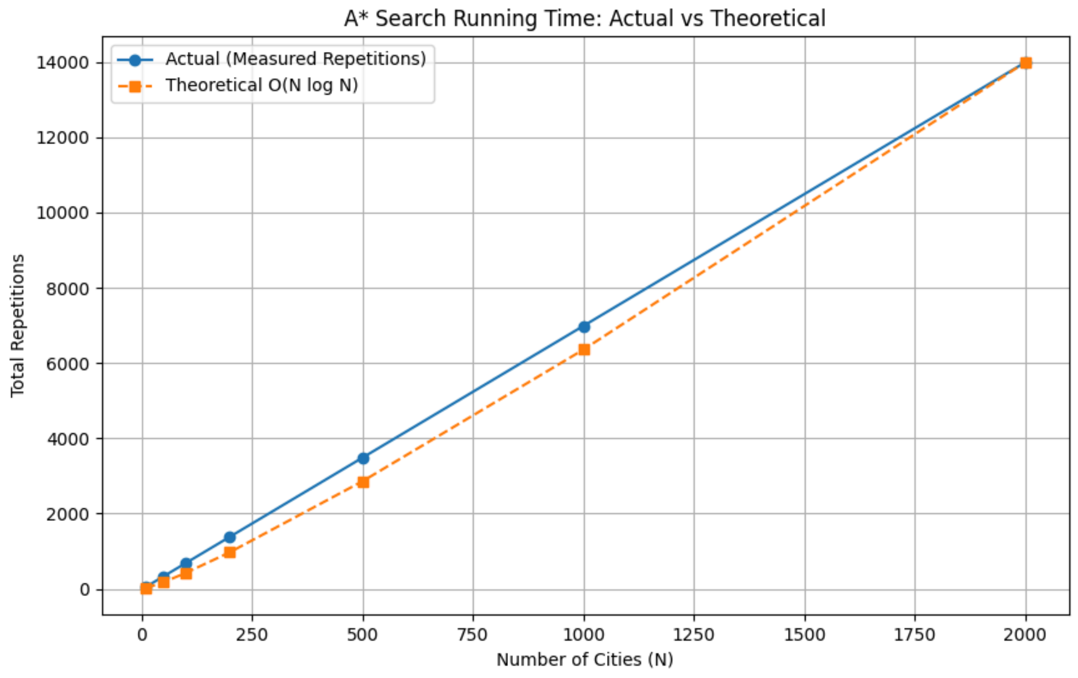
```
N          | Repetitions
------------------------------
10         | 55
50         | 335
100        | 685
200        | 1385
500        | 3485
1000       | 6985
2000       | 13985
```

c - Drawing and Comparing of measured and theoretical running times:

- Imported matplotlib for plotting.
- Calculated Theoretical Data was O((N+E)log N) from previous calculations. Since graph sparse is **(E≅6N)**, the theoretical complexity is roughly **O(NlogN)**.
- Plotted two lines: One for the **measured_data** and one for the **theoretical_data**.

```python
    import matplotlib.pyplot as plt
    import math

    # 1. Theoretical Calculation (O(N log N))
    # We compute a scaling factor 'k' to align the curves visually for comparison
    # Using the largest N (2000) to find k: k = measured_2000 / (2000 * log2(2000))
    last_n = test_values[-1]
    last_meas = measured_data[-1]
    k = last_meas / (last_n * math.log2(last_n))

    theoretical_data = [k * n * math.log2(n) for n in test_values]

    # 2. Plotting
    plt.figure(figsize=(10, 6))

    # Plot Actual (Measured)
    plt.plot(test_values, measured_data, marker='o', label='Actual (Measured Repetitions)')

    # Plot Theoretical
    plt.plot(test_values, theoretical_data, marker='s', linestyle='--', label='Theoretical O(N log N)')

    plt.title('A* Search Running Time: Actual vs Theoretical')
    plt.xlabel('Number of Cities (N)')
    plt.ylabel('Total Repetitions')
    plt.legend()
    plt.grid(True)
    plt.show()
```

Output:



d - Interpreting of results:

Linear Growth of Actual Repetitions:

The blue line ("Actual") is almost perfectly straight (linear). This happens because our counter tracks the number of loop iterations (how many nodes popped + how many neighbors checked).

- Since every node has at most 6 neighbors (|i-j| ≤ 3), the total number of edges (E) is roughly 6N.
- **In this specific problem**, the heuristic **h(i,j) = |i-j| is very weak** compared to the edge weights (w = i+j). **As a result, the A* algorithm does not significantly prune the search space and ends up visiting nearly every node**, **similar to Dijkstra's algorithm**.
- Therefore, the total repetitions are approximately N + 6N = 7N, which is a **linear function O(N)**.

Comparison with Theoretical Curve:

The orange dashed line represents the theoretical complexity O(N log N). It is slightly curved compared to the straight blue line.

- The theoretical curve grows faster than the measured line (in terms of slope change) because it accounts for the cost of Heap operations (heappush/heappop).

- Our code's counter treats every loop iteration as taking "1 unit" of time. In reality, as N grows, the heap operations inside those loops take slightly longer O(log N) to execute.
- Therefore, while the number of repetitions is linear (blue line), the actual CPU time (if measured in seconds) would likely follow the curve of the orange line more closely.

Part 3: Comparison of two algorithms

1. Time Complexity Comparison

Both Dijkstra's Algorithm and A* Search have the same worst-case time complexity when implemented using **a binary min-heap**: **O((N + E) log N).** In this project, the graph is sparse, meaning the **number of edges (E) is proportional to the number of vertices (N)**, specifically **E = O(N)**. Under this condition, the time complexity for both algorithms simplifies to: **O(N log N)**

Measured Performance (Repetitions)

Dijkstra's Algorithm: The measured number of repetitions grew approximately linearly with the number of vertices, that is O(N).

A* Search: The measured repetitions also showed linear growth, nearly identical to Dijkstra's performance, that is O(N).

Reason for Similarity

In theory, A* should outperform Dijkstra by expanding fewer nodes. **However, in this project, the heuristic h(n) was extremely weak** relative to the edge weights w(u, v). Because the **heuristic contributed very little to the evaluation function f(n) = g(n) + h(n)**, **A* was unable to effectively prioritize nodes closer to the goal**. **As a result, it expanded nearly the same nodes in the same order as Dijkstra's algorithm.**

2. Space Complexity Comparison

Both algorithms have very similar memory requirements.

Distance and cost arrays: Both algorithms store distance estimates and predecessor pointers for each vertex, requiring **O(N) space**.

Priority queue: Both maintain a priority queue of discovered nodes. In the worst case, this queue can contain up to O(N) elements.

Heuristic overhead: A* requires a small additional amount of storage and computation for heuristic values, but this **does not change the asymptotic space complexity**.
Overall space complexity for both algorithms is O(N).

3. Advantages and Disadvantages

Advantage of Dijkstra’s Algorithm:
Dijkstra’s Algorithm is an **uninformed search method** that explores the graph uniformly in all directions **without using any domain-specific knowledge**. It is **guaranteed to find the shortest path as long as all edge weights are non-negative**, making it a reliable and robust algorithm. Additionally, Dijkstra’s algorithm is **relatively simple to implement** and **does not require the design of a heuristic function**.

Disadvantage of Dijkstra’s Algorithm:
However, **it is inefficient for point-to-point searches** because it **expands many unnecessary nodes** that lie in directions unrelated to the target.

Advantage of A* Search Algorithm:
A* Search Algorithm, in contrast, **is an informed search technique** that **uses a heuristic to guide exploration** toward the goal. It **guarantees an optimal solution only when the heuristic is admissible**, meaning **it never overestimates the true cost to reach the destination**. When an **effective heuristic is used**, **A* can significantly reduce the number of explored nodes** and **perform much faster than Dijkstra for single-destination queries**.

Disadvantage of A* Search Algorithm:
However, **its performance is highly dependent on the quality of the heuristic**, and **when the heuristic is weak**, **A* degrades to behavior nearly identical to Dijkstra’s algorithm**.

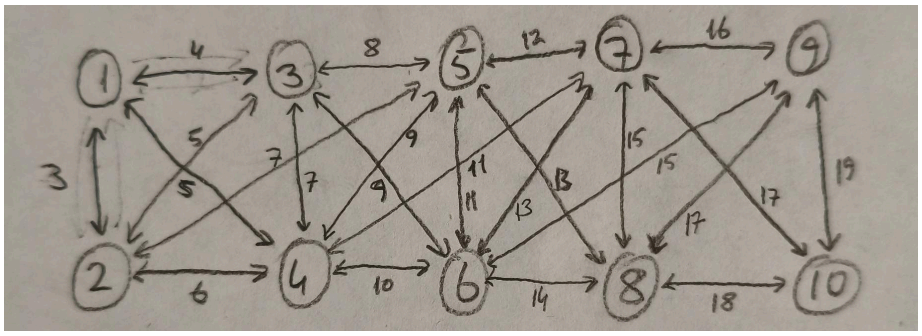
4. Conclusion

In this project, **Dijkstra’s Algorithm and A* Search performed almost identically. Although A* is generally preferred for point-to-point pathfinding, its advantage depends on the quality of the heuristic.** The **heuristic used here was too small** relative to the edge weights to guide the search effectively. **As a result, A* expanded nearly the same number of nodes as Dijkstra’s algorithm**, **leading to similar linear growth in operations**.

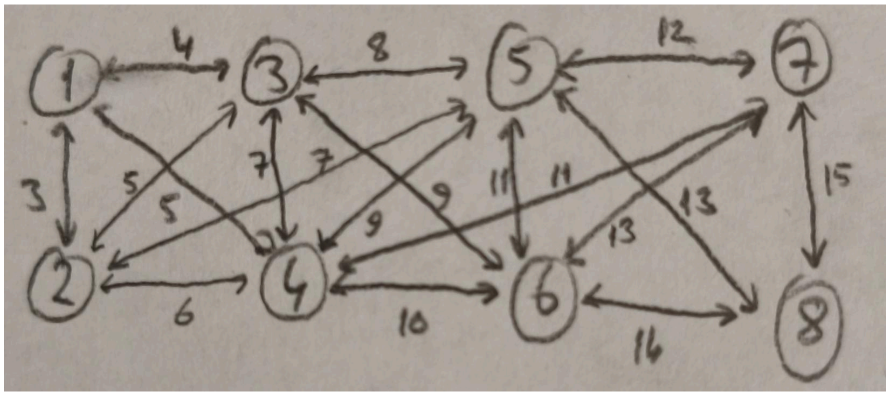
To allow A* to outperform Dijkstra in this context, a more informative heuristic such as scaling the distance estimate by the minimum edge weight would be required.

Part 4: Additional Materials:

The graph below is for N=10 and edges given:



The graph below is for N=8 and edges given:


1
2
3
4
5
6
7
8
4
8
12
3
5
5
7
7
9
9
11
11
13
13
15
6
10
14