**HWK-1 Report**

**Group “Computer Nerds”**

*Riley Boughner 50% | Eli Fouts 50%*

*All team members contributed in equal measure*

*In completing this assignment, all team members have followed the honor pledge specified by the instructor for this course*

**1. Algorithm Implementation**

BFS uses a queue to explore nodes level by level. It stores (city, path, cost) tuples and maintains a visited set. Find solutions with minimum edges but not minimum cost in weighted graphs.

queue = deque([(start, [start], 0)])

city, path, cost = queue.popleft()

queue.append((neighbor, path + [neighbor], cost + edge\_cost))

DFS uses a stack to explore depth-first. Same data structure as BFS but processes nodes in LIFO order. Can get stuck in long paths and doesn't guarantee optimal solutions.

stack = [(start, [start], 0)]

city, path, cost = stack.pop() # LIFO instead of FIFO

**Greedy Best-First** uses a priority queue that always selects the node with lowest heuristic value. Fast but potentially suboptimal since it only considers the heuristic estimate.

fringe = [(heuristic(start), start, [start], 0)]

h, city, path, cost = heapq.heappop(fringe) # Always lowest h

A\* combines actual cost g(n) with heuristic h(n) using f(n) = g(n) + h(n). Uses a priority queue and only marks nodes visited when expanded. Optimal with admissible heuristics.

fringe = [(heuristic(start), 0, start, [start])] # (f, g, city, path)

new\_f = new\_g + heuristic(neighbor) # f = g + h

**2. Heuristic Functions**

Straight-Line Distance Heuristic returns precomputed distances from any city to Bucharest (0-380 range). Admissible because straight-line distance never overestimates road distance. Perfect for Bucharest searches.

def heuristic\_bucharest(city, sld):

return sld.get(city, float("inf"))

# sld = {'Arad': 366, 'Bucharest': 0, 'Pitesti': 100, ...}

Absolute Difference Heuristic calculates |distance\_to\_bucharest(city) - distance\_to\_bucharest(goal)|. Assumes cities with similar Bucharest distances are close to each other. Not admissible - can overestimate and break A\* optimality.

def heuristic\_abs\_diff(city, goal, sld):

return abs(sld.get(city, float("inf")) - sld.get(goal, float("inf")))

# Example: |366 - 253| = 113 for Arad->Sibiu

**3. Performance Analysis**

**A graph with green bars

AI-generated content may be incorrect.A graph with red bars

AI-generated content may be incorrect.**

**A graph of blue bars

AI-generated content may be incorrect.**

The results show significant differences in algorithm performance across three key metrics: computational efficiency, memory usage, and solution quality.

**Time Complexity Analysis**

The nodes expanded metric reveals clear differences in computational requirements. Greedy Best First search proved most efficient, expanding only 4 nodes by following the heuristic directly toward the goal. A\* demonstrated good efficiency with 6 nodes expanded, balancing its need for optimality with heuristic guidance. DFS required 8 node expansions, sometimes following lengthy paths before backtracking to find the solution. BFS performed worst with 9 nodes expanded due to its systematic level by level exploration approach.

**Space Complexity Analysis**

Memory requirements varied considerably between algorithms. Both BFS and DFS maintained small frontiers of 4 nodes maximum, reflecting their simple data structures. Greedy Best First required more memory with a fringe size of 5 nodes due to its priority queue operations. A\* consumed the most memory with 7 nodes in its frontier, as it maintains multiple candidate paths with their associated f-values in the priority queue.

**Solution Quality Comparison**

Path costs revealed the critical trade-off between speed and optimality. A\* found the optimal solution with a cost of 418, guaranteed by its use of an admissible heuristic. BFS produced a suboptimal but reasonable path costing 450, while Greedy Best-First achieved a cost of 447 despite its faster execution. DFS performed poorly with a path cost of 733, demonstrating how its depth-first nature can lead to highly inefficient solutions.

**Algorithm Trade-offs**

The results highlight fundamental trade-offs in search algorithm design. A\* provides the optimal balance for this problem type, guaranteeing the best solution while maintaining reasonable computational and memory costs. Greedy Best-First offers the fastest execution but sacrifices solution quality. BFS and DFS, while conceptually simpler, prove less suitable for weighted graph problems where path cost matters more than path length.

**4. Instructions on running code:**

Pip Install matplotlib

Then run python file

**5. Bibliography**

None