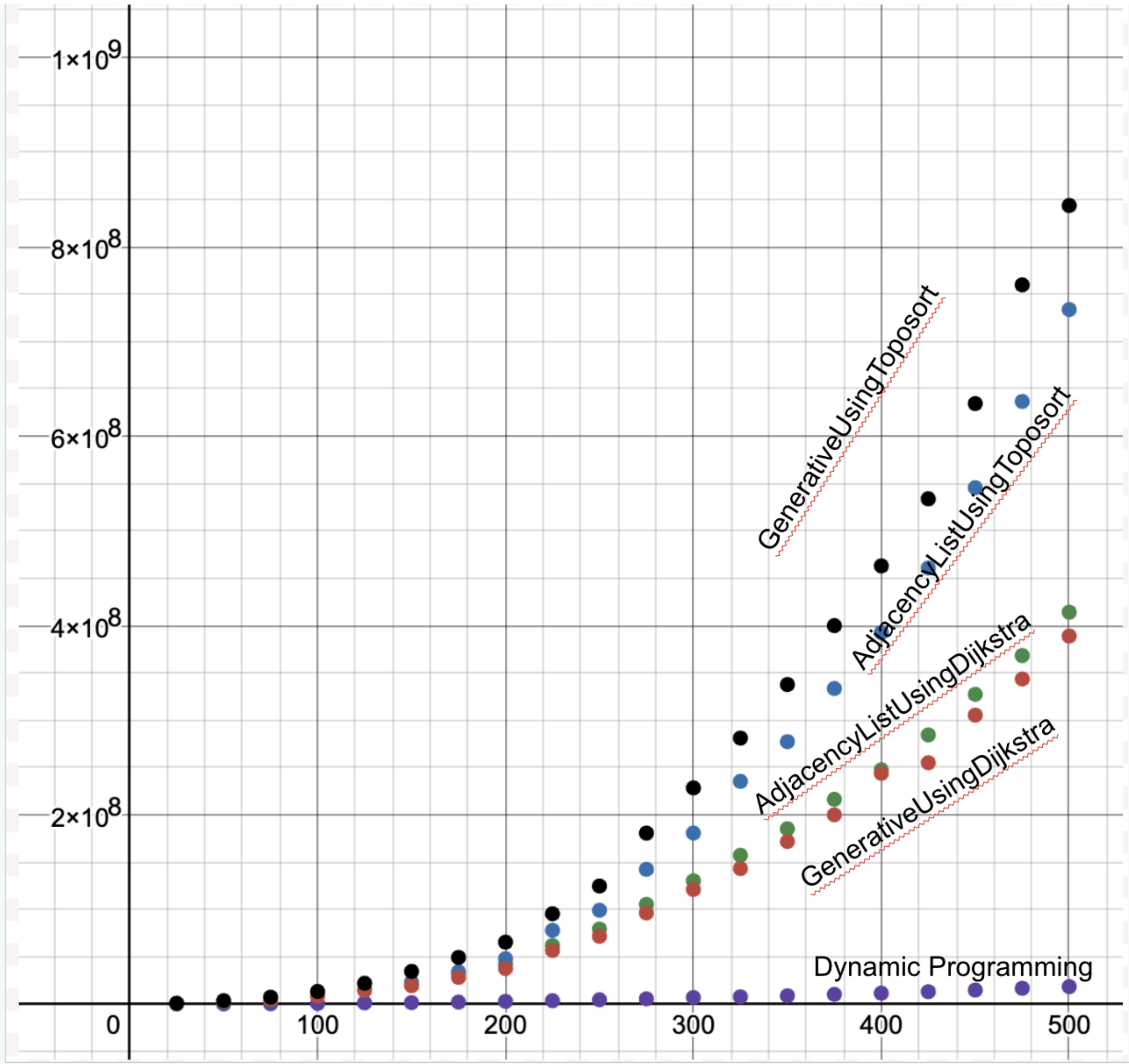
1×10^9
8×10^8
6×10^8
4×10^8
2×10^8
0
100
200
300
400
500
GenerativeUsingToposort
AdjacencyListUsingToposort
AdjacencyListUsingDijkstra
GenerativeUsingDijkstra
Dynamic Programming

Comparison of Runtimes Across All 5 Approaches

1. Dynamic Programming Seam Finder (Purple - Fastest):
    a. Runtime: Fastest overall
    b. Why: Direct computation of seam costs through the dp table with O(N), N as number of pixels W x H, time complexity. The entire graph is traversed once
    c. Graph Traversal:
        i. Traverses the entire graph once to compute the dp table.
        ii. Backtracking adds minimal overhead, making it efficient.
2. Generative Using Dijkstra (Red - Second Fastest):
    a. Runtime: Slightly slower than Dynamic Programming but faster than others
    b. Why:
        i. The generative graph eliminates the need for precomputing
        ii. Dijkstra's solver works efficiently with the adjacency list structure, prioritizing vertices dynamically via the priority queue.
    c. Graph Traversal:
        i. Every pixel is visited and relaxed at least once during edge relaxation. Priority queue operations add log(N) overhead per operation.
3. AdjacencyList Using Dijkstra (Green):
    a. Runtime: Slower than GenerativeDijkstra but faster than topological sort approaches.
    b. Why:
        i. Precomputing the adjacency list speeds up neighbor queries
        ii. The priority queue in Dijkstra's algorithm ensures efficient edge relaxation but adds computational overhead compared to Dynamic Programming.
    c. Graph Traversal:
        i. Each pixel is visited once, but redundant neighbor queries are avoided due to adjacency lists.
4. AdjacencyList Using Toposort DAG Solver (Blue):
    a. Runtime: Slower than Dijkstra-based approaches due to the preprocessing step of computing the topological order.
    b. Why:
        i. Toposort leverages the acyclic nature of the graph but requires DFS for topological ordering before edge relaxation.
        ii. This adds O(N) preprocessing overhead, making it slower for smaller graphs.
    c. Graph Traversal:
        i. First traversal for DFS to compute the topological order.
        ii. Second traversal for relaxing edges in topological order.

5. <u>Generative Using Toposort DAG Solver (Black - Slowest):</u>
    a. Runtime: Slowest
    b. Why:
        i. Computes neighbors dynamically, introducing repeated calculations during topological sorting and edge relaxation.
        ii. The combination of dynamic neighbor generation and topological sorting adds significant overhead.
    c. Graph Traversal:
        i. Traverses the graph at least twice (DFS + relaxation), but dynamic neighbor generation adds extra computational steps.

<u>Impact of SeamFinder and ShortestPathSolver Choices</u>

SeamFinder:

1. DynamicProgrammingSeamFinder:
    a. Fastest when applicable because it avoids any graph abstractions
    b. Operates directly on the input image
2. Generative Seam Finders:
    a. Work well with DijkstraSolve - low memory usage with reasonable performance
    b. Struggle with TopoDAGSolver due to repeated neighbor generation
3. AdjacencyList Seam Finders:
    a. Better suited for solvers that frequently query neighbors, such as DijkstraSolver.
    b. Use more memory but ensure faster neighbor access.

ShortestPathSolver:

4. DijkstraSolver:
    a. Balances computational and memory efficiency.
    b. Works well with both generative and precomputed approaches.
5. TopoDAGSolver:
    a. Best for directed acyclic graphs but less efficient overall due to the need for topological sorting
    b. Generative graphs amplify inefficiency because of repeated neighbor generation.

Graph Traversal Reasons

1. DynamicProgrammingSeamFinder:
    a. Traverses the graph once to fill the dp table.
    b. Uses backtracking to trace the seam, avoiding any graph abstractions.
2. GenerativeDijkstra:
    a. Traverses nodes dynamically as prioritized in the queue.
    b. Minimizes memory use but can increase runtime due to dynamic neighbor generation.
3. AdjacencyListDijkstra:
    a. Traverses each node once.
    b. Benefits from precomputed adjacency lists, avoiding redundant neighbor computations.
4. GenerativeToposort:
    a. Traverses the graph multiple times:
    b. Once for DFS (to compute the topological order).
    c. Again for edge relaxation.
    d. Suffers from repeated neighbor generation during DFS and relaxation.
5. AdjacencyListToposort:
    a. Traverses the graph multiple times but avoids redundant neighbor computations due to precomputed adjacency lists.
    b. Handles DFS and edge relaxation more efficiently than generative approaches.