BFS 1: Map-Based Visited BFS Filename: bfs_01_map_visited.cpp

Source: https://github.com/aashutosh1997/stl-bfs/blob/master/BFS.cc

Key OD: Uses a map (Red-Black Tree) to track visited nodes instead of an array. This changes the visited check from a simple memory load to a complex tree traversal involving dynamic pointer chasing.

BFS 2: Custom Queue and Linked-List BFS Filename: bfs_02_custom_queue.cpp

Source:https://github.com/Eitan02/Graph-Adjacency-List-Implementation/blob/main/Algorithms.cpp

Key OD: Uses a custom fixed-size circular buffer for the queue and manual linked lists for neighbors. The primary difference is the heavy reliance on manual pointer dereferencing during traversal compared to standard vector-based lists.

BFS 3: Full Map-Based BFS (String Keys) Filename: bfs_03_full_map.cpp

Source:
https://github.com/Csaid7/Project-3----BFS-Dijkstra-s-ad-Priority-Queue-Implementation/blob/main/Graph.cpp

Key OD: Relies entirely on maps and string keys for all graph operations. The observed difference is the complete replacement of integer arithmetic with expensive string comparisons and tree traversals for every single operation.

BFS 4: Implicit Grid BFS Filename: bfs_04_grid.cpp

Source: https://github.com/tianruih/Maze-Solver/blob/main/maze.cpp

Key OD: Performs BFS on a 2D grid by calculating neighbor coordinates on-the-fly using arithmetic. This shifts the hardware usage from heavy memory reads to ALU-intensive boundary checks and coordinate addition.

BFS 5: Level Buffered BFS Filename: bfs_05_level_buffered.cpp

Source: https://github.com/AndreasLepik/multi_source_bfs/blob/master/include/textbook-bfs.h

Key OD: Uses two vectors to process nodes in discrete levels, swapping the buffers at the end of each layer. This changes the memory access pattern from a streaming sliding window queue to bulk read and write operations.

BFS 6: Bit-Parallel Multi-Source BFS Filename: bfs_06_bit_parallel.cpp

Source:
https://github.com/AndreasLepik/multi_source_bfs/blob/master/include/bitmapping-msbfs.h

Key OD: Executes up to 64 BFS traversals simultaneously by packing their state into 64-bit integers. It relies heavily on bitwise instructions rather than comparison branches to manage the search state.

BFS 7: 2-Phase Bit Parallel BFS Filename: bfs_07_anp_bit.cpp

Source: https://github.com/AndreasLepik/multi_source_bfs/blob/master/include/anp-msbfs.h

Key OD: Splits processing into two phases: blind aggregation of neighbor bits and then a single state update per node. This significantly reduces random memory reads by checking visited status only once per node per level.

BFS 8: Set-Based Multisource BFS Filename: bfs_08_set.cpp

Source: https://github.com/AndreasLepik/multi_source_bfs/blob/master/include/msbfs.h

Key OD: Simulates concurrent BFS traversals using sets to track which sources have visited a node. It uses complex set algorithms and heavy pointer-chasing logic compared to the simple arithmetic of the Bit-Parallel version.

BFS 9: Deque BFS (0-1 BFS) Filename: bfs_09_deque.cpp

Source: https://github.com/piash76/0-1-bfs/blob/main/0-1bfs.cpp

Key OD: Uses a double-ended queue (deque) to handle weighted edges of 0 or 1. It pushes 0-weight edges to the front and 1-weight edges to the back, testing the specific memory management of the deque container.

BFS 10: Adjacency Matrix BFS Filename: bfs_10_matrix.cpp

Source:
https://github.com/mritunjay11196/BFS-using-Queue-LinkedList-/blob/master/BFS_using_queue_linkedList.cpp

Key OD: Represents the graph as a dense 2D integer array, blindly iterating through every column for every visited node to find neighbors. This forces a highly predictable, contiguous memory scanning pattern that is significantly more expensive than sparse lists.

BFS 11: Standard STL Queue BFS Filename: bfs_11_stlqueue.cpp

Source: Gemini 3 Pro (Generated based on standard implementation pattern)

Key OD: The standard baseline implementation using std::queue and vector-of-lists adjacency structure. Serves as the control group for memory layout comparisons.

BFS 12: Linked List Queue BFS Filename: bfs_12_ll_queue.cpp

Source: Gemini 3 Pro (Generated based on standard implementation pattern)

Key OD: Uses std::list (doubly linked list) as the queue container. Tests node-based heap allocation (scattering nodes in memory) versus contiguous vector/deque allocation.

BFS 13: Priority Queue BFS Filename: bfs_13_priority_queue.cpp

Source: Gemini 3 Pro (Generated based on standard implementation pattern)

Key OD: Uses std::priority_queue instead of a FIFO queue. Changes logic from O(1) push/pop to O(log N) heap operations, drastically changing the instruction trace to include heap rebalancing code.

BFS 14: CSR (Compressed Sparse Row) BFS Filename: bfs_14_csr.cpp

Source: Gemini 3 Pro (Generated based on standard implementation pattern)

Key OD: Stores the entire graph in three compact arrays (row_ptr, col_ind, val). Removes dynamic allocation and pointer chasing entirely during traversal, resulting in the most streamlined memory trace possible.

BFS 15: Pointer-Based Node BFS Filename: bfs_15_pointer.cpp

Source: Gemini 3 Pro (Generated based on standard implementation pattern)

Key OD: Uses a struct Node with real memory pointers rather than integer indices. A hardware fault here is highly likely to result in an immediate Segmentation Fault rather than Silent Corruption.

BFS 16: Bitset Visited BFS Filename: bfs_16_bitset_visited.cpp

Source: Gemini 3 Pro (Generated based on standard implementation pattern)

Key OD: Uses std::vector<bool> (specialized bit-field). Forces the CPU to use bitwise instructions to access single bits of state packed into words.

BFS 17: Integer Distance Array BFS Filename: bfs_17_dist_array.cpp

Source: Gemini 3 Pro (Generated based on standard implementation pattern)

Key OD: Eliminates the boolean visited array entirely. It initializes a distance array to -1 and checks if dist[v] == -1 to determine if a node is new. Combines state tracking with distance calculation.

BFS 18: Unordered Set Visited BFS Filename: bfs_18_uset_visited.cpp

Source: Gemini 3 Pro (Generated based on standard implementation pattern)

Key OD: Uses std::unordered_set (Hash Table) to track visited nodes. Adds hashing logic overhead to every single node visit check.

BFS 19: Bidirectional BFS Filename: bfs_19_bidirectional.cpp

Source: Gemini 3 Pro (Generated based on standard implementation pattern)

Key OD: Runs two simultaneous BFS traversals (Source to Target and Target to Source). The key difference is the intersection check performed at every step, doubling state management logic.

BFS 20: Path-Reconstruction BFS Filename: bfs_20_path_recon.cpp

Source: Gemini 3 Pro (Generated based on standard implementation pattern)

Key OD: Maintains a parent array alongside the visited array. Adds an extra memory write operation for every edge traversal to record the path.

BFS 21: Bottom-Up BFS Filename: bfs_21_bottom_up.cpp

Source: Gemini 3 Pro (Generated based on standard implementation pattern)

Key OD: Inverts logic. Iterates unvisited nodes and checks "Who can reach me?". Completely inverts the memory access pattern from Push to Pull.

BFS 22: Iterative Deepening BFS (IDDFS) Filename: bfs_22_iddfs.cpp

Source: Gemini 3 Pro (Generated based on standard implementation pattern)

Key OD: Simulates BFS using Recursion (Stack) instead of Queue (Heap). Trades high memory usage for high CPU usage (re-visiting nodes) by repeatedly running Depth-Limited Search.

BFS 23: Topological Sort BFS (Kahn's) Filename: bfs_23_topological.cpp

Source: Gemini 3 Pro (Generated based on standard implementation pattern)

Key OD: Adds a constraint where nodes are only added when In-Degree becomes 0. Adds an extra array write (indegree decrement) and zero-check to every step.

BFS 24: Color-Coded BFS Filename: bfs_24_color_coded.cpp

Source: Gemini 3 Pro (Generated based on standard implementation pattern)

Key OD: Tracks 3 states (White, Gray, Black) using an Enum. Adds complexity to the conditional logic compared to a simple binary visited check, altering the branch prediction pattern.

BFS 25: Direction-Optimizing BFS Filename: bfs_25_direction_opt.cpp

Source: https://github.com/sbeamer/gapbs/blob/master/src/bfs.cc

Key OD: Dynamically switches between Top-Down (Queue) and Bottom-Up (Scan) strategies based on frontier size. The gold standard for high-performance traversal with complex heuristic branches.