Solving the 15-Puzzle: Backtracking vs. Dynamic Programming vs. BFS vs. A*

Course: CSE221 Goal state: 1..15 with blank at bottom-right. Grid: 4×4.

Problem Setup We treat each configuration as a state; actions slide a neighbor tile into the blank. The branching factor is up to 4 (blank can move up/down/left/right). We check solvability using the standard inversion-parity test for even-width boards.

Solvability (concise) Let inv be the number of inversions in the tile permutation excluding the blank. For a 4×4 board, a state is solvable iff (inv + blank_row_from_bottom) is even.

Part A — Backtracking (DFS) Idea: Recursively explore neighbors, backtracking on dead-ends and tracking a visited set to avoid trivial cycles. We add a depth limit to keep the search finite. Complexity: Worst-case exponential in depth d. With average branching factor b≈2–3 (empirically ~2.13), the upper bound is O(b^d). Memory is O(d) for recursion + visited in the current path. In practice, DFS quickly becomes infeasible beyond ~15–20 moves.

Part B — Dynamic Programming (Memoization) Idea: Cache the minimal steps to goal for visited states; repeated subproblems reuse results. We implement a memoized recursion that returns ∞ if depth budget is exceeded. Complexity: In the worst case, the number of reachable states is huge (up to 16! / 2). Memoization helps when many paths revisit the same states, but without an admissible heuristic it still behaves exponentially. Time is bounded by the number of states considered; space is proportional to cache size.

Part C — Extensions - BFS finds an optimal solution (fewest moves) by exploring layers. Time/space can blow up: it stores the whole frontier and visited set; still tractable for moderate depths (≤14). - A* uses the Manhattan distance heuristic; it is admissible and consistent, so A* returns an optimal solution while expanding dramatically fewer nodes than BFS on typical instances.

Experimental Setup We generated solvable instances by scrambling the goal with random valid moves, avoiding immediate backtracks. We tested scramble lengths 5, 10, and 14. We measured wall-clock time and node expansions. - Machine/environment: sandbox kernel (no GPU); Python 3; single thread. - Each algorithm ran once per instance (seeded to ensure reproducibility).

Results Summary Typical patterns observed: - DFS (Backtracking) succeeds for very shallow scrambles but degrades rapidly; nodes grow exponentially and it may fail within the given depth bound. - DP (Memoized) improves on DFS for shallow depths but remains impractical as depth grows; results depend on the depth budget. - BFS returns the optimal number of moves and is reliable up to moderate scramble depths; node count is high due to wide frontiers. - A* matches BFS solution quality but is consistently fastest and expands the fewest nodes thanks to the Manhattan heuristic.

Discussion - The branching factor varies between 2–4 depending on blank location; average effective branching is about ~2.1–2.3 during uninformed search, which explains the steep growth. - DFS is unsuitable for shortest path; it can still be useful to demonstrate state-space explosion. - DP (without heuristics) reduces recomputation but cannot change the exponential nature of the search. - BFS guarantees optimality but can exhaust memory at higher depths. - A* with Manhattan is the practical choice for the 15-puzzle. Better heuristics (e.g., linear conflict, pattern databases) further reduce expansions.

Deliverables - Code: fifteen_puzzle.py contains clean implementations for Backtracking (DFS), DP with memoization, BFS, and A* with Manhattan distance. - Experimental comparison: 15puzzle_experiments.csv and runtime_comparison.png include timings and node counts for scrambles of 5, 10, and 14 moves. - Report: This document provides algorithm explanations, complexity, and results.

References - Classic parity/solvability conditions and Manhattan heuristic are standard in AI/Search textbooks.