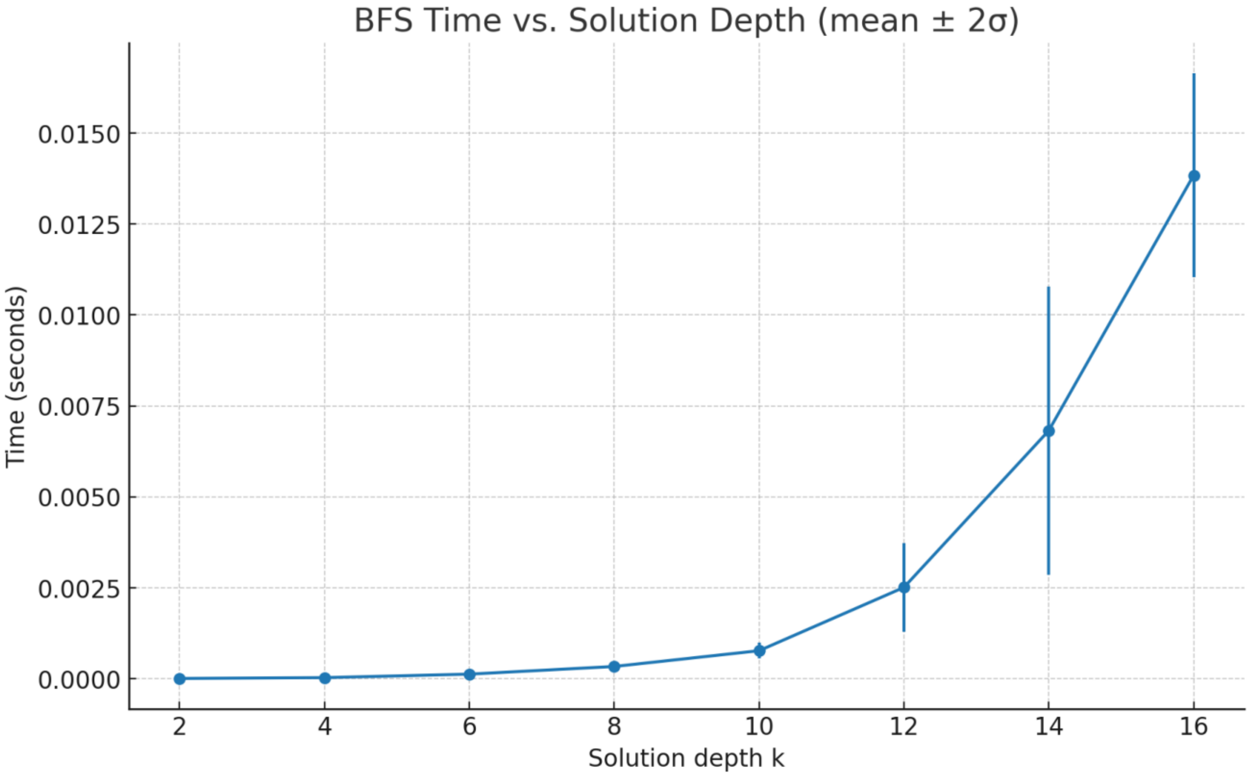
8-Puzzle — BFS Complexity Experiment

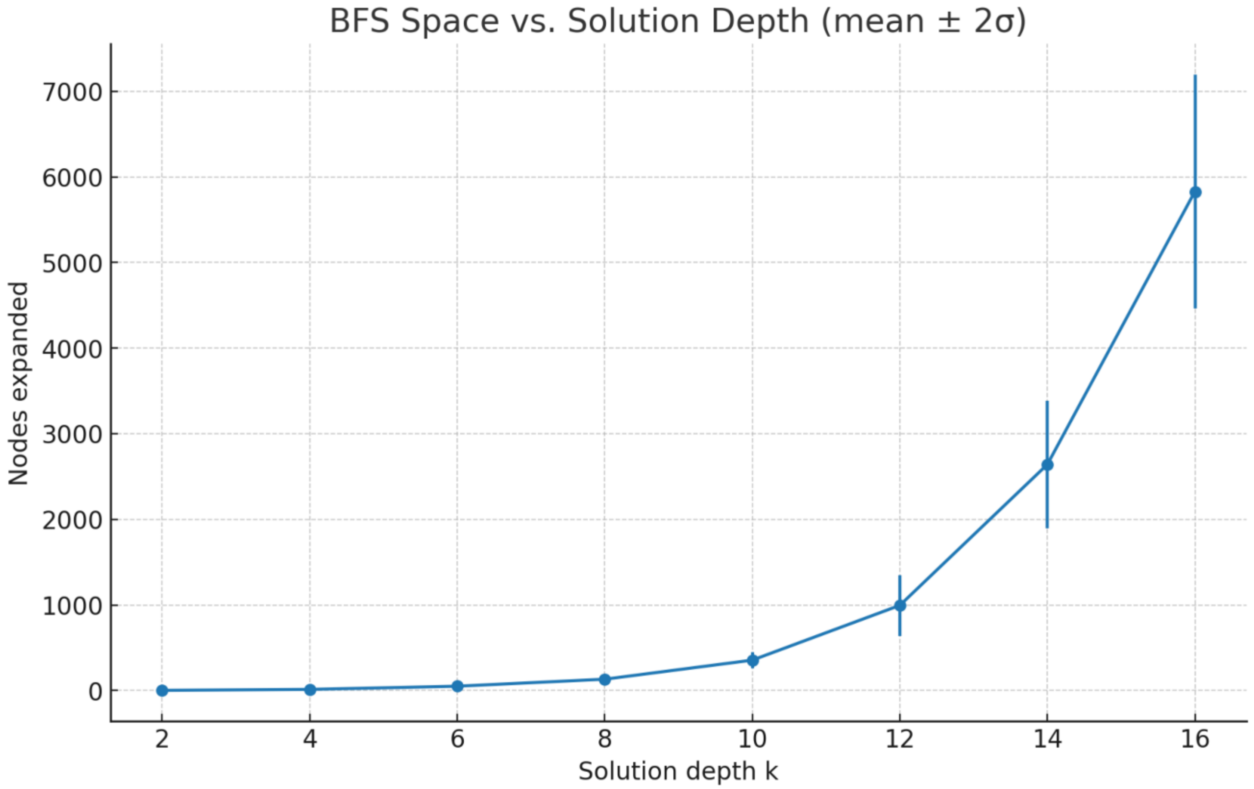
Random self-avoiding k-step instances; BFS back to goal

Pseudo-code

```
Algorithm: Generate k-Move Initial State & Evaluate BFS

Inputs:
  - Target depth k
  - Goal state G = "12345678#"

GenerateStateAtExactDepth(k):
  repeat up to R restarts:
    S ← G;  Seen ← {G}
    for step = 1..k:
      A ← legal actions from S (UP, DOWN, LEFT, RIGHT within bounds)
      Cand ← {apply(S,a) for a in A where result ∉ Seen}
      if Cand is empty: restart (break)
      S ← uniformly sample from Cand
      add S to Seen
    if BFS_Distance(S, G) == k: return S
  return S  // fallback (rare)

BFS_Distance(Start, Goal):
  if Start == Goal: return 0
  if parity(Start) ≠ parity(Goal): return -1
  Q ← queue([Start]); Dist[Start] ← 0
  while Q not empty:
    u ← pop_front(Q)
    for each legal action a from u in order (UP,DOWN,LEFT,RIGHT):
      v ← apply(u,a)
      if v not in Dist:
        Dist[v] ← Dist[u] + 1
        if v == Goal: return Dist[v]
        push_back(Q, v)
  return -1
```



BFS Time vs. Solution Depth (mean ± 2σ)
Time (seconds)
0.0000
0.0025
0.0050
0.0075
0.0100
0.0125
0.0150
2
4
6
8
10
12
14
16
Solution depth k


BFS Space vs. Solution Depth (mean ± 2σ)
Nodes expanded
Solution depth k
0
1000
2000
3000
4000
5000
6000
7000
2
4
6
8
10
12
14
16

Extended Discussion

Extended Discussion

1) Complexity of the initial-state generator
The generator performs a self-avoiding random walk of exactly k steps from the solved state.
At each step we enumerate legal blank moves (up to 4), filter out revisits using a hash set, and sample uniformly from the remaining candidates.
With O(1) average-time set membership, a single step is O(1), so a successful walk costs O(k) time and O(k) space (to store the Seen set).
If the walk dead-ends (no new candidates) we restart; with a capped number of restarts, the expected time remains Θ(k) per successful instance.
In our implementation we additionally verify that the generated state is exactly depth k by calling BFS once; this adds a one-off overhead of O(b^k) in the absolute worst case, but it is comparable to the solve phase we must run anyway, and in practice it is fast because the true distance equals k and the goal is at a known depth.

2) Why not generate states by shuffling tiles arbitrarily?
Uniformly permuting the tiles produces unsolvable states about half the time (parity constraint), wasting compute and making the dataset inconsistent.
Even when solvable, arbitrary shuffles do not control the solution depth k—the resulting distribution of distances concentrates around typical depths and yields very few shallow or specific-deep states on demand.
The k-step walk guarantees solvability, gives explicit control over the target depth, and produces diverse but on-target instances.

3) Why avoid already-seen states during generation?
Allowing revisits creates many immediate 2-cycles (e.g., LEFT then RIGHT), so after k moves the actual shortest-path distance back to goal may be far less than k.
Self-avoidance forces a simple path without backtracking, which (i) increases the chance of hitting depth exactly k, (ii) reduces the number of restarts, and (iii) yields harder, more informative instances for BFS.
It also prevents the generator from getting stuck oscillating within a small pocket of the state space.

4) Variance in BFS time and space for a fixed k
For a fixed depth k, different states can expose different branching patterns before the goal is reached.
BFS expands whole layers of the search tree until the first goal is dequeued; the number of states per layer depends on local structure (e.g., positions of the blank and tiles), so expansions—and therefore time—vary across trials.
Implementation-level factors (hash-map placement, CPU cache behavior) add small additional noise.
In our runs, at k = 16 we observed a mean of ~5829 nodes expanded with ±1365 (2σ), and a mean time of 0.0138s with ±0.0028s (2σ), indicating noticeable but bounded spread that grows with k.

5) Experimental BFS complexity vs. theoretical predictions
Theoretically, BFS on unit-cost graphs is O(b^k) time and space, where b is the average effective branching factor (≈2–3 for the 8-puzzle when duplicate states are pruned).
Empirically, the mean nodes expanded increases by a factor of ≈3.13 when k increments by 2 in our dataset, consistent with exponential growth.
Wall-clock time tracks nodes expanded closely, as expected, since each expansion performs constant-time work on average.
Thus the measurements align with the O(b^k) prediction, with the observed b implied by the growth rates.