1. What happens if you fail to push all successors to the fringe in DFS?

Based on the DFS implementation in `Algorithms.txt`, the `getSuccessors` method provides all possible next states from the current state. If you were to modify this to not push all successors onto the fringe (the stack), you would effectively be **pruning branches of the search tree**. This has two major consequences:

*   **Incompleteness:** The algorithm would no longer be guaranteed to find a solution, even if one exists. If the only path to a goal state goes through one of the successors you failed to push, that path will never be explored.
*   **Non-optimality:** Even if a solution is found, it may not be the best one. A shorter or cheaper path might exist in one of the pruned branches.

In short, failing to push all successors makes the search unreliable and incomplete.

2. Does BFS give fewer actions to reach the goal compared to DFS?

Yes, generally. BFS is guaranteed to find a path with the fewest actions (the shortest path) if all actions have the same cost (e.g., a cost of 1).

Here's why:
*   **BFS (`Algorithms.txt`)** uses a Queue (First-In, First-Out). It explores the search space level by level, meaning it checks all paths of length 1, then all paths of length 2, and so on. The first time it finds the goal, it must be at the shallowest depth, which corresponds to the path with the fewest actions.
*   **DFS (`Algorithms.txt`)** uses a Stack (Last-In, First-Out). It explores as deeply as possible down one path before backtracking. It might find a goal at the end of a very long, winding path before it ever explores a much shorter path that was available closer to the start.

While DFS could get lucky and find the shortest path first, only BFS guarantees it.

3. In the "many paths" scenario, which algorithm has the fewer expansions, and why?

In the autograder results for the `graph_manypaths.test`, **DFS had the fewest expansions.**

Here's the reason:
In a state space with many paths to the goal, DFS's "dive-first" strategy can be more efficient in terms of node expansions. It picks one path and follows it deeply. If that path leads to a goal, it finds a solution without needing to explore the beginning of all the other alternative paths.

BFS, UCS, and A*, on the other hand, are more systematic. At each level, they explore all available options before moving deeper. In a "many paths" scenario, this means they will expand the initial nodes of many different paths, leading to a higher number of total expansions compared to a "lucky" DFS run.

4. Explain how the implementation of A* differs from UCS.

The implementations of A* and UCS in `Algorithms.txt` are very similar, but they differ in one critical aspect: **the priority value used in the priority queue.**

*   **Uniform Cost Search (UCS):** The priority of a node is its path cost from the start state, commonly known as `g(n)`. The code reflects this by pushing nodes with their `new_cost`:
    ```python
    # From UCS implementation
    frontier.push((successor, actions + [action], new_cost), new_cost)
    ```
    UCS always expands the node with the lowest total path cost from the start.

*   **A* Search:** The priority of a node is the sum of its path cost `g(n)` and a heuristic estimate `h(n)` of the cost to reach the goal from that node. This sum is `f(n) = g(n) + h(n)`. The code shows this calculation:
    ```python
    # From aSearch implementation
    priority = new_cost + heuristic(successor, problem)
    frontier.push((successor, actions + [action], new_cost), priority)
    ```
    By incorporating the heuristic, A* guides its search towards the goal, making it more efficient than UCS if the heuristic is well-designed.

5. How many nodes were expanded in Question 7?

Based on the output from running the test for Question 7, **1445 nodes** were expanded.