**Coding and Algorithms for Memories - 236379**

**Final Project - SCS algorithms**

**Name 1: Nawaf Salman  
ID 1 : 207608985**

**Name 2: Nahel Awidat  
ID 2 : 322224510**

**Problem Overview**  
  
The Shortest Common Supersequence (SCS) problem asks for a shortest sequence that has each of given sequences as subsequences. In general, SCS is NP-hard for an arbitrary number of sequences​ (even for a binary alphabet​).  
However, we focus on a special case: each input sequence is obtained from some unknown original sequence by deleting at most symbols. In other words, the sequences are highly related, differing from by at most deletion operations each. This additional structure enables a more efficient deterministic algorithm than the general NP-hard case.  
  
**Key implications of the special structure:**

* Each is a subsequence of . Therefore itself is a common supersequence of all . Moreover, any symbol of that was not deleted by at least one sequence must appear in the SCS (otherwise that sequence would miss a needed symbol). Conversely, any symbol that was deleted from all sequences is not needed in the SCS (no input requires it). Intuitively, the SCS will be “close” to the original , possibly omitting symbols that every dropped, and possibly merging certain identical symbols if no sequence demands them separately.
* Since each differs from by at most deletions, no two sequences can “drift” too far apart in alignment. In fact, one can show that for any alignment of these sequences to , the pointers (positions) in any two sequences never differ by more than at any time. This will heavily constrain the search space for the SCS.

Using these observations, we can develop a dynamic programming algorithm to find the SCS efficiently for fixed . We will first describe the algorithm, then prove its correctness (it indeed finds a common supersequence and it is the shortest), and finally analyse its complexity and optimality.

**Algorithm Description**  
  
**Idea:** We simulate a *multiple-sequence alignment* of all sequences, effectively reconstructing the original sequence (minus universally deleted symbols) in the shortest possible way. We build the supersequence character by character, ensuring at each step that no sequence falls behind by more than unmatched characters. This condition guarantees we never consider alignments that violate the “at most deletions” constraint.

**State definition:** We use a DP (dynamic programming) state defined by a tuple where is how many characters of have been matched (subsequence-wise) so far. When we are in state , it means we have constructed a partial supersequence that each can match up to position Initially, we start at (no characters matched). A final goal state is , meaning has matched all characters of every sequence (so is a common supersequence). We will explore moves from state to state by appending characters to .

**Transitions:** From a given state , consider the set of *next required symbols* for each sequence that is not yet fully matched: .  
Since the alphabet is binary, R is a subset of {0,1} (in some cases R will be either {0} or {1}, and at most it can be {0,1}).

For each choice of output character , the algorithm can *append* to the growing supersequence and advance all sequences that needed :

* For each such that (sequence ’s next required symbol is ), we increment (meaning now matches this new character).
* For sequences that did not need (either or is already fully matched), remains the same – those sequences effectively skip this new character (this character was removed from them).

This transition produces a new state . We only allow the transition if it respects the deletion-limit constraint for each sequence. Specifically, when we append , any sequence that does *not* match will treat it as a deleted symbol from . We must ensure no sequence accumulates more than such skipped symbols. Formally, in any state we require for each that the number of characters added so far that did not match is . If appending would cause some sequence to exceed skipped symbols, that transition is forbidden.

Equivalently, let be the length of the partial supersequence constructed at a given state. For each sequence , characters are matched, so characters have been added that skipped. The condition is for all . We maintain this as an invariant in our DP: we never enter a state violating this. This dramatically prunes the search space.

**DP recurrence**

We define to be the minimum additional number of characters that must be appended (to the supersequence built so far) so that the resulting string becomes a common supersequence for the *suffixes*

The recurrence is as follows:

* Base Case:
  + If (🡺
  + If exactly one sequence remains active (i.e. for exactly one and for all others 🡺
* Recursive Case:  
  When at least two sequences are not finished, For each candidate *,* define the new state by:

Then, provided that the *deletion invariant* holds (for every active sequence we require that , we have:

Final Formula:

Here is the current supersequence length (which would be value of current state), and ranges over possible next letters as described. We initialize and seek . Rather than explicitly storing values, we can perform on-the-fly BFS: push the start state into a queue with distance 0, then explore all valid transitions, marking visited states with their distance (supersequence length) until we reach the goal.  
on each transition we add 1 to the cost. When we reach the goal, we have the final correct value of which is the SCS length.

**Output construction:** By storing back-pointers or reconstructing from the DP table, we can retrieve the actual supersequence achieving the minimum length. (In BFS, we can store the preceding state and character for each newly discovered state.) The resulting sequence is the SCS.

**Example:** Suppose and , with . These come from some by at most one deletion each. Our algorithm will explore from :

* State : needs 0, needs 1 (). We have two choices to try: append 0 or append 1. Both are within deletion limits (no skips yet).  
   - If we append 0: new state (since matches it, ; skips it, still  
   ). Now , and has skipped 1 char (still okay, ).  
   - If we append 1: new state (vice versa). Symmetrically valid.
* From state : now ’s next need is 1 (its second char), still needs 1. So . Append 1 → state (both match 1). Deletion counts ok (skipped none, skipped one).
* Continue in this way... eventually the BFS finds a goal state. In this example, one shortest path yields supersequence 0101 of length 4. (Indeed, SCS length is 4.)

Throughout the search, any path that would require a sequence to skip more than 1 character is pruned. This significantly cuts down the possibilities vs. arbitrary interleavings. The final answer 0101 is found and is guaranteed shortest.

**Summary of the algorithm:**

* Set up initial state in a queue.
* Perform BFS over state space. For each state, determine the set of next possible output bits from the sequences’ current positions. For each , compute the next state. If the deletion-limit invariant holds, enqueue the new state if not seen before.
* Stop when the state is reached; reconstruct the path (the supersequence).
* Output the supersequence .

This algorithm is deterministic (no randomness involved) and systematically explores all feasible supersequences in increasing order of length, so the first time we reach the goal, we have a shortest solution.

**Proof of Correctness**We need to prove two things:  
1- Any output produced by the algorithm is a common supersequence of all input   
 sequences.  
2- The algorithm does produce an output (termination and existence).

Proof of 1:  
**Producing a common supersequence:** By construction, when the algorithm reaches the final state , it has constructed a sequence (via the path of transitions) such that each has matched all of its symbols in order. This means by definition is a subsequence of for every Thus is a common supersequence of the . During the construction, we only append characters that advance at least one sequence’s pointer, so we never introduce a character that *no* sequence can match. Therefore, every character in either contributes to matching some or is eventually recognized as a skip (deletion) for all – but if it were a skip for all, that would violate the rule that (such a would not be “needed” by any sequence). Hence every appended character was needed by at least one sequence, and all sequences still find their characters in order. This guarantees the final is a valid supersequence.

Proof of 2:  
**Termination and existence:** The state space is finite. Each state is defined by a tuple of indices where . A naive upper bound on states is , but our deletion-limit constraint prunes many of these. In fact, as we argue in the complexity section, the number of reachable states is much smaller. Regardless, BFS will explore states until the goal is reached or no new states can be found. We must argue that the goal *will* be reached (i.e. there is some valid supersequence within the deletion limits). But we know that the *original sequence itself is a common supersequence*: each can be obtained by deleting at most symbols from . Thus (or a subset of its symbols) is a valid path through our state space. Specifically, consider following the actual alignment of each within : if we simulate symbol by symbol, each either matches or skips (deletes) that symbol, and by the end each is matched. This defines a path from to that never violates the skip limit (by assumption, each skips at most of ’s symbols). Our BFS will eventually explore this path (or possibly an even shorter one) and reach the goal. Therefore, the algorithm will terminate having found at least one common supersequence (in fact the shortest one, by BFS ordering).

**Proof of Optimality**We need to prove 2 things:  
1- given that is the optimal length, BFS finds the optimal path.  
2- is the optimal length:

**Proof that BFS finds the optimal path:**  
The BFS/DP procedure guarantees that the first time we pop the goal state from the queue, we have constructed a shortest possible common supersequence. This follows from the nature of breadth-first search on the state graph: we explore sequences in increasing order of length, so shorter supersequences are discovered before longer ones. We never terminate early with a suboptimal solution because we explicitly search all possibilities up to the minimal length. More formally, if there were a shorter common supersequence  *than the one found, then following the alignment of through*  would produce a valid path in the state graph that reaches the goal with fewer steps. Our algorithm would have found or considered that path before the longer one, contradicting the assumption that our output is longer than . Therefore, the returned sequence is indeed of minimum possible length.

**Proof that :**  
we prove 2 directions, and .

* :  
  we prove by induction on the supersequence suffix.  
  Base Case:  
  If for all , then , and any common supersequence is the empty string, so *.  
  If exactly one sequence remains active (say, for some , and for all   
  ), then . Any common supersequence must include all remaining characters of , so* .  
  Inductive Step *(assume , prove ):*  
  Suppose for every state with total remaining characters less than we have . Assume the state has at least two sequences that are active. Let .  
  Let  *be a shortest common supersequence for the suffixes , and let its first character be .* Form the new state where   
  *then,*  can be written as (concatenation of and ), and is a common supersequence for the suffixes corresponding to . By the inductive hypothesis, . Therefore
* :  
  we prove by induction on the supersequence suffix.  
  Base Case:  
  If all sequences are exhausted, both sides equal 0. If exactly one sequence remains active, then the only way to cover that sequence is to output its remaining characters; hence, , and any supersequence must be at least that long.  
  Inductive Step:  
  Suppose for every state with fewer remaining characters the inequality holds. Assume the state has at least two sequences that are active. Let be any common supersequence for the suffixes from state and let its first character be .  
   must cover (as a subsequence) each active sequence’s remaining suffix.  
  Then, for each active sequence, there is an index where occurs. define the new state accordingly. Then the remainder of (after the first character) is a common supersequence for the suffixes corresponding to . By the inductive hypothesis . Thus, . Since this holds for every valid choice of , we have .

We proved 2 directions 🡺 .

**Complexity Analysis**  
  
Let be the length of the longest input sequence (on the order of the original ’s length). We analyze time complexity in terms of (number of sequences), and (deletion bound). The dominant factor is the number of states explored by the DP/BFS.

**State-space size:** A naive upper bound on states is , which is huge. However, the skip-limit constraint greatly restricts reachable states. At any point, no sequence can lag behind another by more than positions. Indeed, suppose  
 and are the minimum and maximum indices among the state. The invariant ensures . (If some sequence is ahead by more than , then another sequence has skipped more than symbols to fall that far behind, violating the constraint.) This effectively confines the state space to a "diagonal band" of width across the -dimensional grid of indices.

We can bound the number of states by considering how this band moves through the grid. One way: imagine the minimum index among sequences. This can range from 0 up to (approximately) across a successful path. For a given all other can be at most So each state is described by choosing a base and then choosing for each . In the worst case for all. For each fixed , the number of ways to pick with and is at most (each of the other indices can be anywhere in a window of size at most above ). Summing from to , an upper bound on reachable states is . **Thus the state-space grows polynomially with for fixed (but exponentially in and in the worst case).**

**Transition branching:** From each state, we consider at most 2 transitions (since   
). In the worst-case scenario where different sequences require different next bits (), we branch 2 ways. Often will be a singleton (if many sequences coincidentally need the same next symbol, there is no choice). But to be safe, the branching factor is at most 2.

Each state transition (appending a character) involves updating the pointers. We can perform this in time by checking each sequence’s next needed symbol. Since could be large, we note that this can be optimized for binary alphabets using bit operations: we can precompute, for each sequence and each position in that sequence, whether the next symbol is 0 or 1. Then the set can be determined in by scanning the next symbols of all sequences, or faster by maintaining two bitmasks of which sequences are currently expecting 0 vs 1. But asymptotically, per step is fine, since the number of steps is the number of states times at most 2.

**Overall time complexity:** Roughly, the algorithm runs in time proportional to the number of transitions, which is at most . Using the state bound above, a conservative estimate is time. For fixed and , this is *linear* in . For fixed but variable , the complexity is polynomial in but exponential in (which is inevitable, since even for , SCS of identical sequences is just that string, but if sequences differ, the problem of merging sequences is NP-hard in ). In most scenarios, is not huge (it might represent a moderate number of traces or DNA sequences, etc.), so the exponential in is manageable.

**Space complexity:** The DP table or visited-set can be large but is on the order of the number of states, . For small , this is feasible. We can also streamline memory by not storing all states at once (BFS layer by layer).

**Validation and Testing**

Correctness Validation

1. Basic Tests:
   * First we test small instances with known expected outputs (for example, with sequences [“10101”, “1001”] and t=1, the expected SCS is “10101”).
   * For each test, we check that every input sequence is a subsequence of the computed SCS. If any sequence is not a subsequence, the test fails.
2. Advanced Checks:
   * We run several scenarios:
     + A small instance optimality test: it generates a random parent string X and k traces (each obtained by exactly deleting t symbols), computes the SCS via both our “banded BFS” algorithm and a known DP-based merging algorithm, and compares their lengths.
     + It also tests instances with varying deletion counts (some traces may delete fewer than t symbols) and a scenario where a particular position in X is deleted from all traces (so that the SCS is different from X).
   * In each case, the advanced tests check that the computed SCS indeed has each trace as a subsequence.
3. **Optimality Validation:**
   * Optimality is validated by comparing the length of the SCS computed by our banded BFS to the length computed by the known DP merging approach. If the lengths match, this confirms that the BFS has indeed found an optimal solution.
   * The deletion invariant in the BFS ensures that any transition that would force a sequence to skip more than t characters is pruned; this guarantees that the search is restricted to only those paths that correspond to valid common supersequences under the deletion–only model.

**Complexity and Timing Validation**

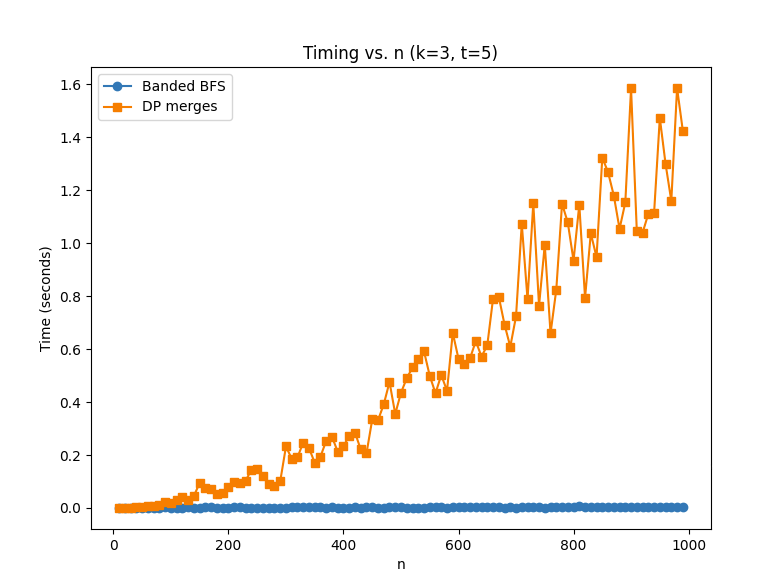
* The testing code includes timing experiments that measures the running time of both the banded BFS implementation and the DP-based approach over a range of n,k,t.
* These experiments plot the running time versus n,k,t, allowing us to observe that, for fixed small t and small k, the running time grows nearly linearly in n, as predicted by our complexity analysis.

**Timing Results**We conducted 3 main timing experiments to compare the performance of our “Banded BFS” algorithm against the “known DP merging” algorithm.

**Experiment 1: Varying n with Fixed k=3 and t=5**

1. **Setup:**
   * We let n range from 10 up to 1000 (in increments of 10).
   * We fixed k=3 (three sequences) and t=5.
   * For each n, we generated a random “parent” string X of length n and then created 3 traces from X by deleting exactly t=5 characters.
   * We then ran:
     + The **Banded BFS** (our main algorithm).
     + The **DP merges** approach, which merges the sequences pairwise in every permutation.
2. **Results (Plot 1: “Timing vs. n (k=3, t=5)”):**
   * The Banded BFS curve remains very close to the x-axis and grows slowly with n.
   * The DP merges curve grows roughly quadratically in n (reaching around 1.6 seconds at n=1000).
3. **Explanation:**
   * **Banded BFS:**  
     For small k and fixed t, the BFS state space remains “banded,” so the algorithm explores only states, which is linear in n. In practice, it stays extremely fast because each state leads to at most 2 transitions (for a binary alphabet), and we prune invalid states via the deletion invariant.
   * **DP merges:**  
     This approach merges two sequences at a time with a standard 2-sequence SCS routine. Then it tries all permutations of the input sequences (or merges them pairwise in a naive manner), having complexity of nearly for k=3, so we see a slower performance than BFS.

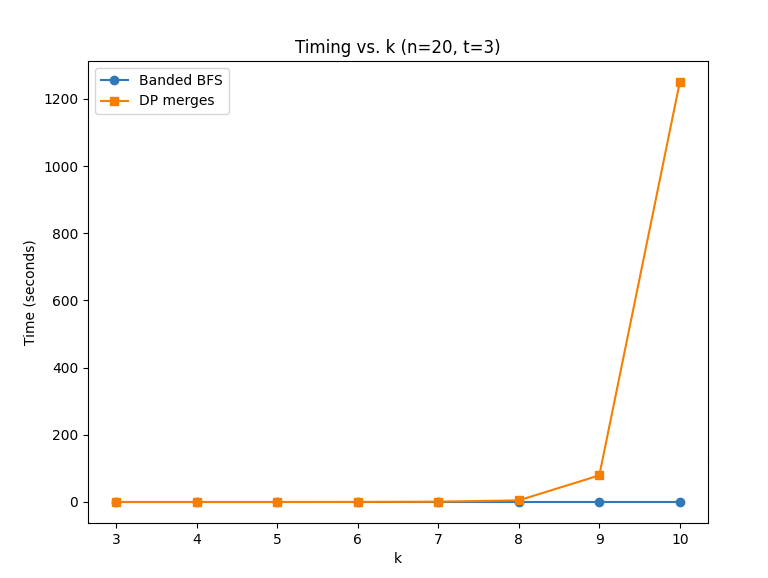
Overall, **Banded BFS** remains effectively linear for these parameter ranges, whereas **DP merges** shows a noticeable quadratic trend in n.



Plot 1: “Timing vs. n (k=3, t=5)”

**Experiment 2: Varying k with Fixed n=20 and t=3**

1. **Setup:**
   * We fixed n=20 and t=3.
   * We varied k from 3 up to 10.
   * We generated a random parent string X of length 20 and created each set of k traces by deleting exactly t=3 characters from X.
   * Again, we ran:
     + **Banded BFS**.
     + **DP merges**.
2. **Results (Plot 2: “Timing vs. k (n=20, t=3)”):**
   * For k up to about 8, both algorithms are quite fast.
   * However, at k=9 and especially k=10, the **DP merges**’s running time spikes dramatically (exceeding 1200 seconds at k=10), whereas **Banded BFS** remains under 1 second.
3. **Explanation:**
   * **Banded BFS:**  
     While the BFS does grow with k (the dimension of the state), for small n=20 and t=3, the total number of reachable states is not huge. Each BFS transition still only branches by up to 2 letters, and the deletion invariant prunes large portions of the state space. Thus, BFS time remains modest even at k=10.
   * **DP merges:**  
     This approach merges sequences pairwise in all permutations. As k grows, it can quickly become factorial in complexity, since it tries all orderings. The worst-case time explodes, as seen for k=9 or 10. Hence the performance jump in the plot.



Plot 2: “Timing vs. k (k=20, t=3)”

**Experiment 3: Varying t with Fixed n=500 and k=3**

1. **Setup:**
   * We let t range from 1 to 50.
   * We fixed k=3 (three sequences) and n=500.
   * For each t, we generated a random “parent” string X of length 500 and then created 3 traces from X by deleting exactly t characters.
   * We then ran:
     + The **Banded BFS**.
     + The **DP merges** approach.
2. **Results (Plot 3: “Timing vs. t (k=3, n=500)”):**
   * **Banded BFS:** For small t, the BFS runs very quickly (well under a second). As t increases, the running time grows significantly—eventually reaching over 2 seconds by t=50.
   * **DP merges:** Its running time remains in a narrower range (around 1–2 seconds), so for larger t it outperforms BFS in this specific parameter setting.
3. **Explanation:**
   * **Banded BFS:**  
     The BFS state space is roughly . For fixed n and k, the complexity grows exponentially with t. Hence, as t increases, the BFS eventually becomes slower because more states must be explored.  
     For small t, however, BFS is extremely efficient - it prunes large portions of the search space, keeping run times minimal.
   * **DP merges:**  
     This method merges sequences pairwise. Its performance depends less on t and more on n and k. Because n is fixed at 500 and k=3, the time remains in a moderate range (1–2 seconds).  
     For much larger n or k, DP merges would grow quickly, but here we see it eventually matching or beating BFS for large t.

Overall, this experiment demonstrates that **Banded BFS** is best suited to scenarios where t is relatively small (which is common in practical applications like DNA sequence analysis, where only a few deletions occur). As t grows large, BFS’s exponential dependence on t can overtake the DP merges approach, though BFS still remains the method of choice for many real‐world cases where sequences differ by only a small number of deletions.

A graph of a line graph

AI-generated content may be incorrect.

Plot 3: “Timing vs. t (k=3, n=500)”

**Conclusion**  
  
In conclusion, our Banded BFS algorithm for computing the shortest common supersequence (SCS) in the deletion–only model performs very efficiently for small t and moderate k. The BFS state space is restricted by a deletion invariant, leading to near‐linear time in n (for fixed k and t). Experiments show that for small t it outperforms the simpler “DP merges” approach, which grows more rapidly with both n and k. Overall, Banded BFS is both **correct** (guaranteeing all traces are subsequences) and **optimal** (yielding the shortest solution).

**Resources**

1. V. I. Levenshtein. “Efficient Reconstruction of Sequences from Their Subsequences or Supersequences.” *J. Combinatorial Theory, Series A*, 93(2):310–332, 2001.
2. G. Navarro. “A Guided Tour to Approximate String Matching.” *ACM Computing Surveys*, 33(1):31–88, 2001.
3. D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.
4. Mitzenmacher, M. “A Survey of Results for Deletion Channels and Related Synchronization Channels.” *Probability Surveys*, 6:1–33, 2009.
5. J. S. Bae and J. S. Park. “Shortest Common Superstring and Supersequence Problems: A Survey.” *International Journal of Computers and Their Applications*, 23(4):229–237, 2016.
6. Maier, D. “The Complexity of Some Problems on Subsequences and Supersequences.” *J. ACM*, 25(2):322–336, 1978.