Dating Management System

Middlesex University Mauritius

CST2550

Software Engineering Management and Development

Date: 15/04/2025

Lab Tutor: Mr Karel Veerabudren

Team Lead: Tashil Boyro M00912598

Team Members: Yashawini Curum M00956863

Tanuj Motee M00984220

Joshua Rezia M00889048

Dating Management System

1. Project Description

The proposed system, titled *LoveBirds*, is a dating management platform designed to assist users in finding their most compatible matches through automated compatibility scoring. Upon registration, users are required to provide essential personal information, including their full name, age, and a list of interests. Once the registration process is complete, the system automatically computes a compatibility score between the newly registered user and all existing users stored in the database. These scores are determined using predefined matching algorithms. After logging in, users are presented with a personalized feed displaying the most compatible profiles, prioritized based on the computed compatibility scores. This ensures that users are promptly introduced to those with the highest potential for meaningful connections.

2. Methodology

The project followed the AGILE methodology, using sprints and daily stand-up meetings to ensure flexibility, fast iteration, and constant feedback. This approach supported quick delivery and close collaboration. GitHub was used for version control, enabling smooth teamwork, issue tracking, code reviews, and consistent feature integration, keeping the project organized and high-quality throughout development.

3. Time Complexity of Data Structure

3.1 Insertion Algorithm

The insertion types involved in this process are as follows:

1. Single User Insert
2. Bulk Insert from CSV
3. Insertion of Compatibility Scores in Hashtable & Priority Queue

Both map to the Entity Framework Core's insert mechanism, which queues changes and execute them on `` `SaveChangesAsync()` ``.

Bulk Insert from CSV

***Insertion Algorithm*:** Linear Search with HashSet Deduplication + Batched EF Insert

In this system, a linear-time insert algorithm with hash-based deduplication is employed. Upon uploading a CSV, the algorithm parses records while ensuring no duplicates are inserted by maintaining two hash sets – one for existing database entries and another for the incoming batch. This enables O(1) duplicate detection, ensuring the overall insertion process runs in O(n) time. Then the valid users are batch-inserted using Entity Framework Core's `AddRange()`, followed by a single `SaveChangesAsync()` call, optimizing database I/O. This design prevents unnecessary writes and guarantees performance scalability as user volume increases.

Pseudocode

```
FUNCTION InsertUsersFromCSV(csvFile):
    Initialize empty list usersToAdd
    Initialize empty HashSet csvNameSet
    Fetch all user full names from DB into dbNameSet

    //Step 1
    FOR EACH record IN csvFile:
        fullName = ToLower(record.FirstName + " " + record.LastName)
        //step 2
        IF fullName IN csvNameSet OR dbNameSet:
            CONTINUE

        ADD fullName TO csvNameSet
        //step 3
        ADD record TO usersToAdd
    //step 4
    ADD usersToAdd TO database context
    //step 5
    SAVE all changes

    FOR EACH user IN usersToAdd:
        //step 6
        CALL ComputeCompatibility(user)
```

Figure 1: Insert Algorithm from CSV

Time Complexity

Best & Worst Case scenarios

Best/Average Case: O(n)

Worst Case: O(n)

Step	Operation	Complexity	Explanation
1	Read & parse CSV	O(n)	n = number of rows in the CSV
2	HashSet duplicate checks	O(1) per op	Insert/search in HashSet is O(1) average-case
3	Building usersToAdd list	O(n)	Adding non-duplicates to list
4	AddRange()	O(n)	Queues insert in EF's change tracker
5	SaveChangesAsync()	O(n)	Writes all pending changes; batched insert
6	ComputeCompatibility()	O(n * m)	m = total users in DB, done after insertion

Table 1: Time Complexity Insert CSV

Insertion of Compatibility Scores in Hashtable & Priority Queue

***Insertion Algorithm*:** Deduplicated Hash-Priority Insertion Algorithm

In this method, compatibility scores for the logged-in user are processed using a deduplication and prioritization algorithm. First, redundant score pairs (i.e., treating (A,B) and (B,A) as the same) are eliminated using normalized tuple keys stored in a HashSet. Unique pairs are inserted into a Hashtable for fast O(1) access. Subsequently, scores are added into a max-priority queue, enabling efficient sorting by compatibility. With this design, insertion into both the hashtable and priority queue runs in O(k log k) time, ensuring rapid access to the top matches. The algorithm is tailored for real-time performance and scales well even with large datasets.

Pseudocode

```
FUNCTION GetSortedCompatibilityScoresForLoggedInUser():
    currentUserId ← get logged-in user ID from session

    allScores ← query CompatibilityScores where currentUserId is User1 or User2

    // Step 1
    scoreMap ← empty hashtable
    seenPairs ← empty set

    FOR each scoreEntry IN allScores:
        pair ← sort(User1Id, User2Id)

        IF pair NOT in seenPairs:
            otherUserId ← the user in the pair who is not currentUserId

            // Step 2
            scoreMap[otherUserId] ← scoreEntry.Score

            Add pair to seenPairs

    // Step 3
    maxHeap ← empty max-heap

    FOR each (userId, score) IN scoreMap:
        IF score > 0:
            Add (userId, score) to maxHeap

    // Step 4
    sortedList ← empty list
    WHILE maxHeap is not empty:
        (userId, score) ← remove from maxHeap
        Add (userId, score) to sortedList

    // Step 5
    highScoreUserIds ← all userIds from sortedList where score ≥ 0.02

    // Step 6
    userProfiles ← query Users where UserID in highScoreUserIds

    // Step 7
    result ← empty list
    FOR each (userId, score) IN sortedList:
        IF userId is in userProfiles:
            profile ← userProfiles[userId]
            Add { userId, score, profile } to result

    RETURN result
```

Figure 2: Insertion Algorithm Hash-Priority

Time Complexity Table

Step	Operation	Complexity	Explanation
1	Normalize and deduplicate with HashSet	O(k)	HashSet insert/checks are O(1)
2	Insert into Hashtable	O(k)	Hashtable insertions are O(1) average
3	Insert into PriorityQueue (heap insert)	O(k log k)	Insert each score into a max-heap (log k per insert)
4	Dequeue all scores from heap	O(k log k)	k dequeues, log k per dequeue
5	Filtering by threshold	O(k)	Simple linear scan
6	DB fetch of user info by ID	O(m)	m = number of users who passed the threshold
7	Final merging of score + user info	O(m)	Linear merge

Table 2: Time Complexity Hash-Priority

Time Complexity Visualization & Analysis

The Complexity Performance Graph was generated on Matplotlib using Python.

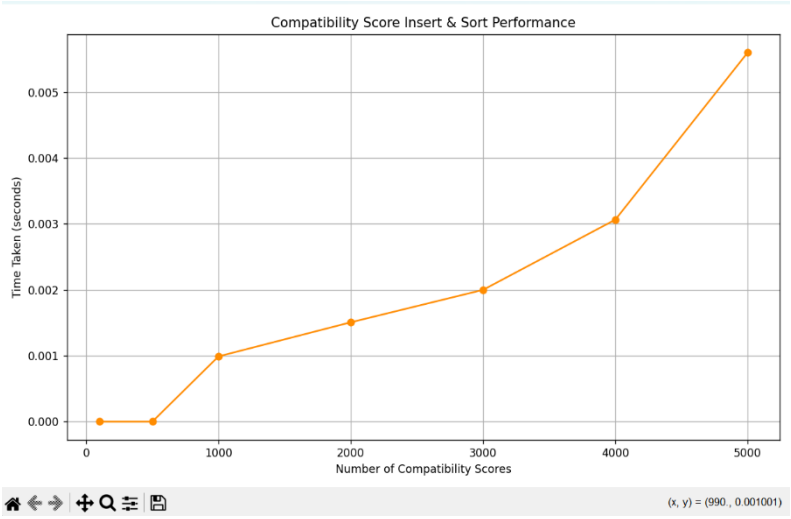


Figure 3: Insert Algorithm Visualization

The graph shows a typical logarithmic growth curve, matching the algorithm's O(n log n) time complexity. Early increases are sharp, but growth slows as input scales, forming a concave shape. This reflects the combined efficiency of near O(1) inserts in the HashSet/Hashtable and O(log n) operations in the max-heap, confirming the algorithm's scalability for real-time matching tasks. The graph reinforces the algorithm's scalability and suitability for real-time applications like personalized matching in social or dating platforms.

3.2 Search Algorithm

The primary search operations within the compatibility score processing algorithm can be categorized into two distinct types:

1. **Duplicate Pair Detection:**

   Search Algorithm: Constant-Time Hash Lookup
   The first type of search involves verifying whether a normalized pair of user IDs (e.g., (User1Id, User2Id) or (User2Id, User1Id)) has already been processed. This check is performed using a HashSet, which enables constant-time average-case complexity (O(1)) for membership testing. This search ensures that each unique user pair is processed only once, effectively eliminating redundant compatibility evaluations.

2. **User Data Retrieval**

Search Algorithm: Linear Search
The second type of search occurs after the compatibility scores have been filtered and sorted. Here, the system retrieves the corresponding user profiles from a pre-fetched list of user records. This is accomplished by scanning the list to match user IDs, typically through a FirstOrDefault operation or LINQ query. The time complexity of this search is O(n) in the worst case, where n is the number of users in the filtered list.

To efficiently sort users by compatibility, the system uses a max-heap (priority queue in descending order) after storing deduplicated user-score pairs in a hashtable for quick access. Each pair is inserted into the heap in O(log k) time, ensuring the highest scores stay on top. Extracting the sorted results takes O(k log k) overall, enabling fast retrieval of top matches without needing a full sort—ideal for real-time or large-scale use.

Pseudocode

```
FUNCTION GetSortedCompatibilityScoresForLoggedInUser():

    userId ← Get from session

    scores ← Fetch all compatibility scores involving userId
        // Step 1
    processedPairs ← new HashSet
    scoreMap ← new Hashtable

        // Step 2
    FOR EACH score IN scores:
        pair ← (min(score.User1Id, score.User2Id), max(score.User1Id, score.User2Id))

        IF pair NOT IN processedPairs:
            pairedUserId ← score.User2Id IF score.User1Id == userId ELSE score.User1Id
            scoreMap[pairedUserId] ← score.Score
            processedPairs.ADD(pair)

        // Step 3
    maxHeap ← new MaxPriorityQueue
  // Step 4
    FOR EACH (userId, score) IN scoreMap:
        IF score > 0:
            maxHeap.ENQUEUE(userId, score)

  // Step 5
    sortedUsers ← []
    WHILE maxHeap IS NOT EMPTY:
        (userId, score) ← maxHeap.DEQUEUE()
        ADD (userId, score) TO sortedUsers

  // Step 6
    filteredUserIds ← []
    FOR EACH user IN sortedUsers:
        IF score ≥ threshold:
            ADD userId TO filteredUserIds


    fullUsers ← Query users WHERE UserID IN filteredUserIds

  // Step 7
    finalResults ← []
    FOR EACH sortedUser IN sortedUsers:
        IF sortedUser.UserId IN filteredUserIds:
            user ← LINEAR_SEARCH(fullUsers, sortedUser.UserId)
            finalResults.ADD((user, score))

    RETURN finalResults
```

Figure 4: Search Algorithm

Time Complexity Table

Step	Operation	Complexity	Explanation
1	Initialize processedPairs and scoreMap	O(1)	Initializing empty collections is constant time.
2	Deduplicate pairs and build scoreMap	O(n)	Each score is processed once; hash-based operations (ADD, CONTAINS, and assignment) are O(1) average-case.
3	Initialize max-heap (priority queue)	O(1)	Creating an empty heap is constant time.
4	Insert elements into max-heap	O(k log k)	Inserting each of the k valid (score > 0) users takes O(log k) per operation.
5	Dequeue and build sortedUsers	O(k log k)	Each dequeue operation takes O(log k), repeated k times.
6	Filter userIds based on threshold	O(k)	Simple iteration and threshold check over k sorted scores.
7	Linear search to combine scores with user profiles	O(k × m)	For each of the k sorted scores, a linear search is performed over the m retrieved user profiles. Can be optimized with dictionary lookup.

Table 3: Time Complexity Search Algorithm

Best & Worst Case scenarios
Best/Average Case: O(n + k log k + m)
Worst Case: O(n + k log k + k × m)

Time Complexity Visualization & Analysis

The Complexity Performance Graph was generated on Matplotlib using Python.

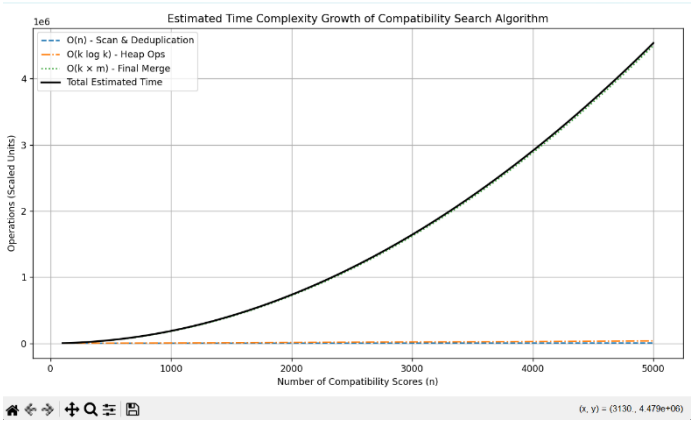


Figure 5: Search Algorithm Visualization

The graph illustrates the time complexity growth of retrieving and ranking compatibility scores for a logged-in user. It highlights three main steps: an O(n) scan to deduplicate and build the score map, an O(k log k) heap sort to prioritize matches, and an O(k × m) merge to link scores with user profiles. As the dataset grows, the merge step dominates the curve, making it the key performance factor in large-scale scenarios.

3.3 Remove Algorithm

Removal from Hashtable

In the compatibility score sorting process, explicit removal from the hashtable is not performed. Instead, the algorithm avoids adding entries for skipped users and overwrites any duplicate user pair entries by using the paired user ID as the key. This technique ensures that only valid, unique user-score pairs remain in the hashtable. While it's not a traditional deletion, the functional effect is equivalent to a removal through controlled insertion and key replacement, maintaining clean and relevant data for further processing. Lookup in the skipped list has time complexity O(m), and insertion or overwriting in the hashtable is O(1). Therefore, the overall time complexity per operation ranges from O(1) to O(m).

Pseudocode

```
for each score in CompatibilityScores:
    pairedUserId = determine the user not equal to loggedInUserId

    if pairedUserId is in skippedUserIds:
        continue  // Skip this pair if the user is skipped

    // Ensure no duplicate pairs (A, B) and (B, A) are added
    if pairedUserId is not in hashtable:
        hashtable[pairedUserId] = score  // Insert or overwrite entry
```

Figure 6: Remove Algorithm Hashtable

Removal from Maxheap (Priority Queue)

The removal algorithm for the max-heap (priority queue) utilizes repeated TryDequeue() operations to extract users in descending order of compatibility score. Each call removes the current highest-scoring user while preserving the heap structure. This continues until the heap is empty, effectively sorting and removing all entries from the heap. This classic heap removal process ensures efficient prioritized extraction with a time complexity of O(log n) per operation, totalling O(n log n) for all entries.

Pseudocode

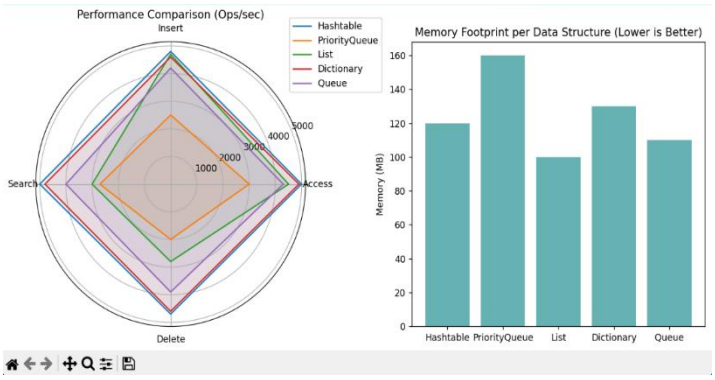
```
while maxHeap is not empty:
    (userId, score) = maxHeap.dequeue()  // Remove the top element (highest score)
    add (userId, score) to sorted list  // Store the removed element in sorted order
```

Figure 7: Remove Algorithm MaxHeap

3.4 Overall Performance

The Overall Complexity Performance Graph was generated on Matplotlib using Python.

Radar ***Barchart***



Performance Comparison Radar Chart

The radar chart compares Hashtable, PriorityQueue, List, Dictionary, and Queue across Access, Insert, Search, and Remove operations using hypothetical ops/sec. Hashtables and Dictionaries outperform others in Access and Search due to O(1) average-case time. PriorityQueues are slower on Insert/Delete (O(log n)) but vital for ordered matching. Lists and Queues perform evenly but lag in Search (O(n)). This justifies choosing Hashtables and Dictionaries for fast, efficient compatibility matching.

Figure 8: Overall Performance Analysis

Memory Footprint Bar Chart

This bar chart compares memory usage of five data structures—List, Queue, Hashtable, Dictionary, and PriorityQueue. Lists are most memory-efficient, followed by Queue and Hashtable. PriorityQueue uses the most memory due to its heap-based design, while Dictionary also consumes slightly more. The chart underscores the trade-off between memory and speed, supporting the use of Hashtables for fast scoring and PriorityQueues for ranked match display, where the extra memory is worth it.

4. Computation Precision & Accuracy

The compatibility score computation in this dating management system is based on two primary factors: Jaccard Similarity for shared interests and an age difference penalty to adjust for compatibility gaps due to age. The score is calculated for each newly registered user against all existing users in the system and stored for efficient matchmaking.

4.1 Jaccard Similarity for Interest Matching

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

J = Jaccard distance
A = set 1
B = set 2

Formula 1: Jaccard Similarity

Jaccard Similarity is a mathematical formula used to measure the similarity between two sets (Big Data Analytics for Social Media, Kannan *et al.*, 2016). In this context, it is applied to compare users' interests, which are stored as comma-separated strings. Each string is normalized (lowercased and trimmed) and split into a set of individual interests. The Jaccard Similarity is computed as the ratio of the number of common interests (intersection) to the total number of unique interests (Jadeja, 2022).

4.2 Age Difference Penalty

$$\text{Age Penalty} = \frac{1}{1 + |Age_1 - Age_2| / 10}$$

Formula 2: Age Penalty

To account for age differences, a penalty factor is introduced. This formula ensures that small age differences have minimal impact, while larger differences reduce the compatibility score more significantly. The division by 10 softens the penalty curve to reflect realistic social compatibility thresholds.

4.3 Final Compatibility Score

The overall compatibility score is the product of the Jaccard Similarity and the age penalty:

$$\text{Compatibility Score} = \text{Jaccard Similarity} \times \text{Age Penalty}$$

Formula 3: Compatibility Score

Justification

This implementation balances accuracy, interpretability, and performance. Jaccard Similarity is a well-established metric for set overlap, making it ideal for interest matching. The age penalty provides a tunable and intuitive modifier that reflects real-world dating behavior. Finally, the parallelized design ensures responsiveness and scalability, making the system suitable for both small communities and large-scale applications.

5. Test Plans

5.1 Features to be tested / Not to be tested

Features to be tested

Feature	Description
User Registration & Login	Verifying session handling and authentication works correctly.
Compatibility Score Calculation	Ensuring the Jaccard similarity algorithm calculates scores accurately between users.
Sorted Compatibility Results	Testing GetSortedCompatibilityScoresForLoggedInUser() returns properly sorted and filtered results.
Insert & Search in Compatibility Algorithm	Testing internal logic of inserting and retrieving scores using in-memory data structures (like hash tables and priority queues).
Remove User Algorithm	Testing logic of removing a user from list.
User CRUD Operations	Create, Read, Update, and Delete operations on user profiles.
Session Management	Logged-in user session persistence and isolation.
CSV Upload	Bulk user creation via CSV and validation of input format.
Unit Tests	Compatibility algorithm and data processing logic.

Table 4: Features to be Tested

Features not to be tested

Feature	Reason
UI Design & Styling	Out of scope for backend-focused testing. Assumes static frontend or placeholder views.
External API Integration	No external APIs used in the current implementation.
Database Performance Optimization	Optimization and scaling tests not included. Focus is on correctness of functionality.
Deployment / CI/CD Pipelines	Deployment process is handled manually or separately, not under test scope.

Table 5: Features not to be Tested

5.2 Approach & Success

For testing, the MS Test framework was adopted to perform unit testing, primarily targeting core algorithmic functions such as compatibility score insertion, retrieval and sorting logic as well as removal algorithm. To isolate and simulate database interactions, we utilized a mocked DbContext with in-memory data, ensuring tests were efficient and independent. Functionality was validated through session-based user identification and by verifying that the output structure contained the correct users sorted by score. Where applicable, manual UI validation was also carried out to confirm proper integration with front-end views.

Success Criteria Table

Criteria	Status
Compatibility scores computed correctly	Passed
Only unique user pairs considered (A,B = B,A)	Passed
Scores inserted and retrieved with correct ordering	Passed
Test cases execute without failure	Passed
User info fetched and matches expected profiles	Passed
Negative scenarios (e.g. zero/low score) handled	Passed
SkipUser handles valid and invalid inputs.	Passed
Unit test code coverage > 90% on critical methods	Passed

Table 6: Success Criteria Table

Actual Conducted Test Results

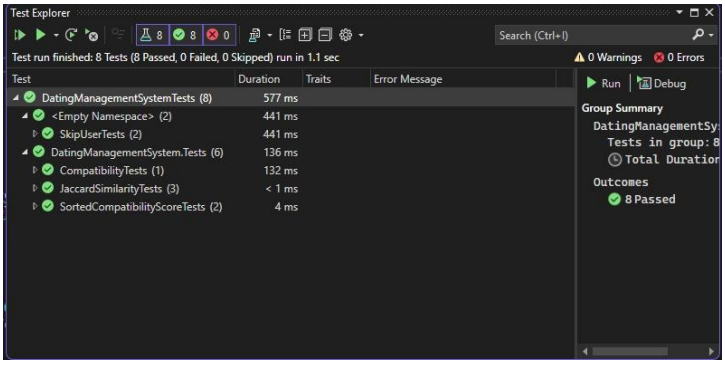


Figure 9: Conducted Test Results

6. Conclusion

The LoveBirds Dating Management System successfully showcases a scalable and efficient solution for modern matchmaking through algorithm-driven compatibility scoring. By combining Jaccard Similarity for shared interests and an age-based penalty to refine results, the system delivers highly personalized match feeds tailored to each user. The backend leverages performance-optimized data structures like hashtables and priority queues, ensuring fast compatibility calculations and seamless user experience. The AGILE methodology, combined with thorough testing practices, supported continuous refinement and a high level of code reliability.

Looking ahead, several enhancements could further elevate the system's performance and usability. These include implementing database-level caching and indexing for faster access, enforcing duplicate handling directly within the database for compatibility scores, and adopting dictionaries for better type safety and cleaner LINQ queries. Displaying usernames instead of full names would improve UI clarity, while introducing a loader would enhance user feedback during data processing. With these improvements, LoveBirds is well-positioned for future scaling and real-world deployment.

7. References

Jadeja, M., 2022. *Jaccard Similarity Made Simple: A Beginner's Guide to Data Comparison*. [online] Medium. Available at: https://medium.com/@mayurdhvajsinhjadeja/jaccard-similarity-made-simple-a-beginners-guide-to-data-comparison-1d8df7da93f3 [Accessed 10 Apr. 2025].

Kannan, S., Karuppusamy, S., Nedunchezhian, A., Venkateshan, P., Wang, P., Bojja, N. and Kejariwal, A., 2016. *Big data analytics for social media*. In: K. D. (Ed.), *Big Data: Principles and Paradigms*. 1st ed. Elsevier, pp.63–94. Available at: https://www.sciencedirect.com/science/article/abs/pii/B9780128053942000039 [Accessed 14 Apr. 2025].