Cse 150 Assignment 1 Report

Ketan Kelkar (krkelkar@ucsd.edu, A12045566)
Soham Jatakia (sjatakia@ucsd.edu, A11555559)
Ajay Bhullar (apbhulla@ucsd.edu, A10624774)

- Description of the problem and the algorithms used to solve problems 1 - 5.
- Describe the data structure used in each algorithm.

- Description of the problem:
  Given an initial prime number and a final prime number, the goal is the find a path comprised of prime numbers such that between any single transition, exactly one digit is varied. The solutions make use of several different search methods.

1. Breadth first search:

Start with the initial prime. Compute all possible configurations of the initial prime which differ by exactly one digit. This is done by iterating through the digits of the initial prime and substituting all digits 0-9. Add all prime configurations to a queue (first in first out). Repeat procedure by exploring and removing items from queue one at a time until the final prime is reached or the queue is empty (i.e. all reachable primes have been explored). Already visited nodes are added to a dictionary and excluded from the queue.
The queue is a first in first out, that means that the first element added to the queue is the first one that will be taken out, it is exactly like the data is waiting in line. We also used a dictionary to store the explore set, a dictionary is basically a hashtable with constant lookup times. Data is stored in a [key,value] pair.

2. Depth limited search:

Start with initial prime. Compute all possible configurations of the initial prime which differ by exactly one digit. This is done the same way done in problem 1. Add all prime configurations to a stack (last in first out). Repeat procedure by exploring and popping configurations one at a time from the stack until the final prime is reached or all reachable primes at a specified depth level have been explored. The depth level is controlled by using recursion. There is no ‘stack’ data structure, but rather the stack frames created through recursion are used.

The stack frames emulate a LIFO stack through virtue of the order they are created in, when they resolve and pop the last frame to be put on the stack is the first frame to be removed.

3. Iterative deepening depth first search:

The same procedure as above, except the limit on the depth of the search is incremented until success or every reachable configuration is explored. No additional data structures are used.

4. Bidirectional search:

The same procedure as the breadth first search is followed except both initial and final prime configurations are considered as starting nodes to create two different queues. The visited nodes set are created individually for each queue, and once an overlap is discovered, both paths are registered. As a result, each path lead up to a point where it can use the other path to finish the link. The same data structures as breadth first search are used.

5. Heuristic search:

Starting with the starting prime, all reachable configurations are computed. Each configuration is assigned a cost based on a heuristic and added to a priority queue. Further explorations of configurations are done in order of the priority queue. We have two versions of heuristics. The first is the hamming function which compares the current configuration to the goal configuration and assigns cost based on similarity between the two configuration when they are treated as strings and path cost thus far. The other one takes the cost to be the sum of all the sums of each digit. This heuristic chooses numbers further away from the current one and the path cost thus far. Our testing suggests that this is not an optimal heuristic. A priority queue is a queue that orders the entries by the minimum element(in python, it can also order it by maximum depending on the implementation). A get() will pop off the element with the lowest cost(or highest depending on the implementation).

- Perform some analysis on the efficacy of the different algorithms with different puzzles. For instance, you can try this problem for large prime numbers and for different sets of prime numbers. You can then measure the number of nodes visited / maximum size of the queue / time it took to run the code/ path length given by algorithm etc., for each case. How do they compare against what you expect from the big-O analysis? How do they compare against each other?

- Results of the analysis and a short discussion. It should include at least one graph with proper labels that shows how the quantity you measured changes with what you varied.

  Conceptually, the breadth first search returns the shortest length path because it searches for the solution at uniform depth similar to the iterative deepening depth first search. The depth limited search is worse on average because it goes all the way down a single path until it fails and then is forced to backtrack. The heuristic search is the most efficient because it prioritizes which paths to explore based on the current configuration's proximity to the goal.
  For all the searches, the branching factor scales based on the number of digits in the input numbers because every extra digit increases the available state space.
  The expected big-O complexity of breadth first search is O(b^d), where b is the branching factor and d is the depth of the solution.
  The expected big-O complexity of depth limited search is O(b^l) where b is the branching factor and l is the limit.
  The expected big-O complexity of the iterative deepening depth first search O(b^d) where b is the branching factor and d is the depth of the solution. This time complexity is closer to that of breadth first search because like in breadth first search, all possible paths of smaller length are checked before those of larger lengths.
  The expected big-O complexity of the heuristic A^* search is O(b^d) where b is the branching factor and d is the depth of the solution. The branching factor here however also depends on the effectiveness of the heuristic function in choosing certain branches to explore over others.

- Test case 1: 103 199
  - Problem 1:
    - Output: 103 193 199
    - Timing: 71 ms
  - Problem 2:
    - Output: 103 503 523 593 193 199
    - Timing: 60 ms
  - Problem 3:
    - Output: 103 193 199
    - Timing: 27 ms
  - Problem 4:

        - Output: 103 109
          199 191
        - Timing: 28 ms
    - Problem 5a:
        - Output: 103 109 199
        - Timing: 30 ms
    - Problem 5b:
        - Output: 103 109 199
        - Timing: 26 ms

- Test case 2: 1009 4969
    - Problem 1:
        - Output: 1009 8009 8069 8969 4969
        - Timing: 785 ms
    - Problem 2:
        - Output: 1009 5009 8009 8069 8969 4969
        - Timing: 59 ms
    - Problem 3:
        - Output: 1009 8009 8069 8969 4969
        - Timing: 181 ms
    - Problem 4:
        - Output: 1009 8009 8069
          4969 4909 2909
        - Timing: 51 ms
    - Problem 5a:
        - Output: 1009 1019 4019 4919 4969
        - Timing: 35 ms
    - Problem 5b:
        - Output: 1009 1069 2069 2969 4969
        - Timing: 31 ms

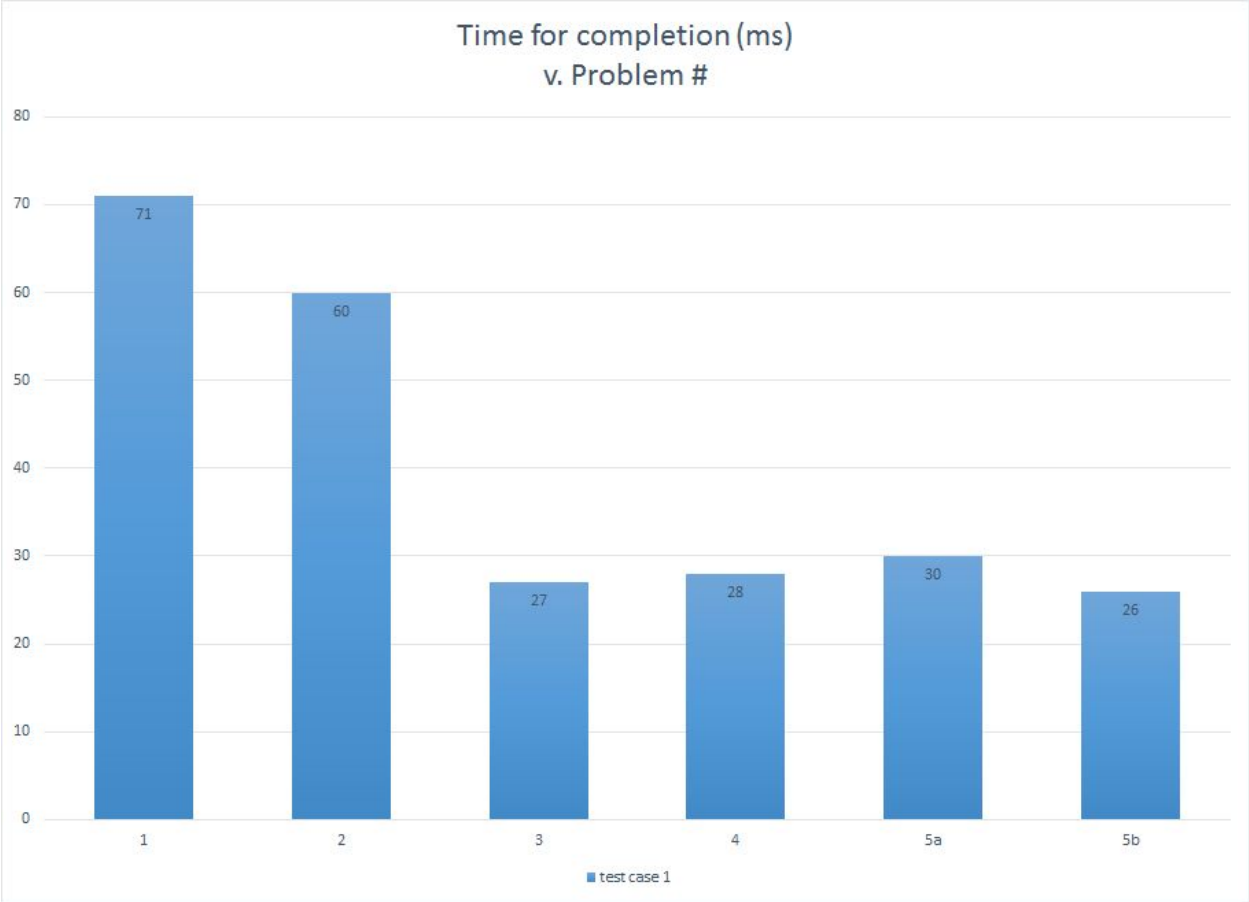
- Test case 3: 103 1231
    - Problem 1:
        - Output: UNSOLVABLE
        - Timing: 52 ms
    - Problem 2:
        - Output: UNSOLVABLE
        - Timing: 582 ms
    - Problem 3:

        - Output: UNSOLVABLE
        - Timing: 2 min 43 sec 307 ms (163,307 ms)
    - Problem 4:
        - Output:
        - Timing:
    - Problem 5a:
        - Output: UNSOLVABLE
        - Timing: 251 ms
    - Problem 5b:
        - Output: UNSOLVABLE
        - Timing: 263 ms

- Test case 4: 11657 26449
    - Problem 1:
        - Output: 11657 61657 65657 65647 65447 65449 66449 26449
        - Timing: 29 sec (29,000 ms)
    - Problem 2:
        - Output: UNSOLVABLE
        - Timing: 11.69 sec (11,690 ms)
    - Problem 3:
        - Output: 11657 61657 65657 65447 66449 26449
        - Timing: 3 min 19 sec 949 ms (199949 ms)
    - Problem 4:
        - Output:
        - Timing:
    - Problem 5a:
        - Output: 11657 10657 10457 10477 20477 20479 26479 26449
        - Timing: 113 ms
    - Problem 5b:
        - Output: 11657 16657 16057 16007 16001 16061 16069 16369 16349 16249 26249 26449
        - Timing: 49 ms

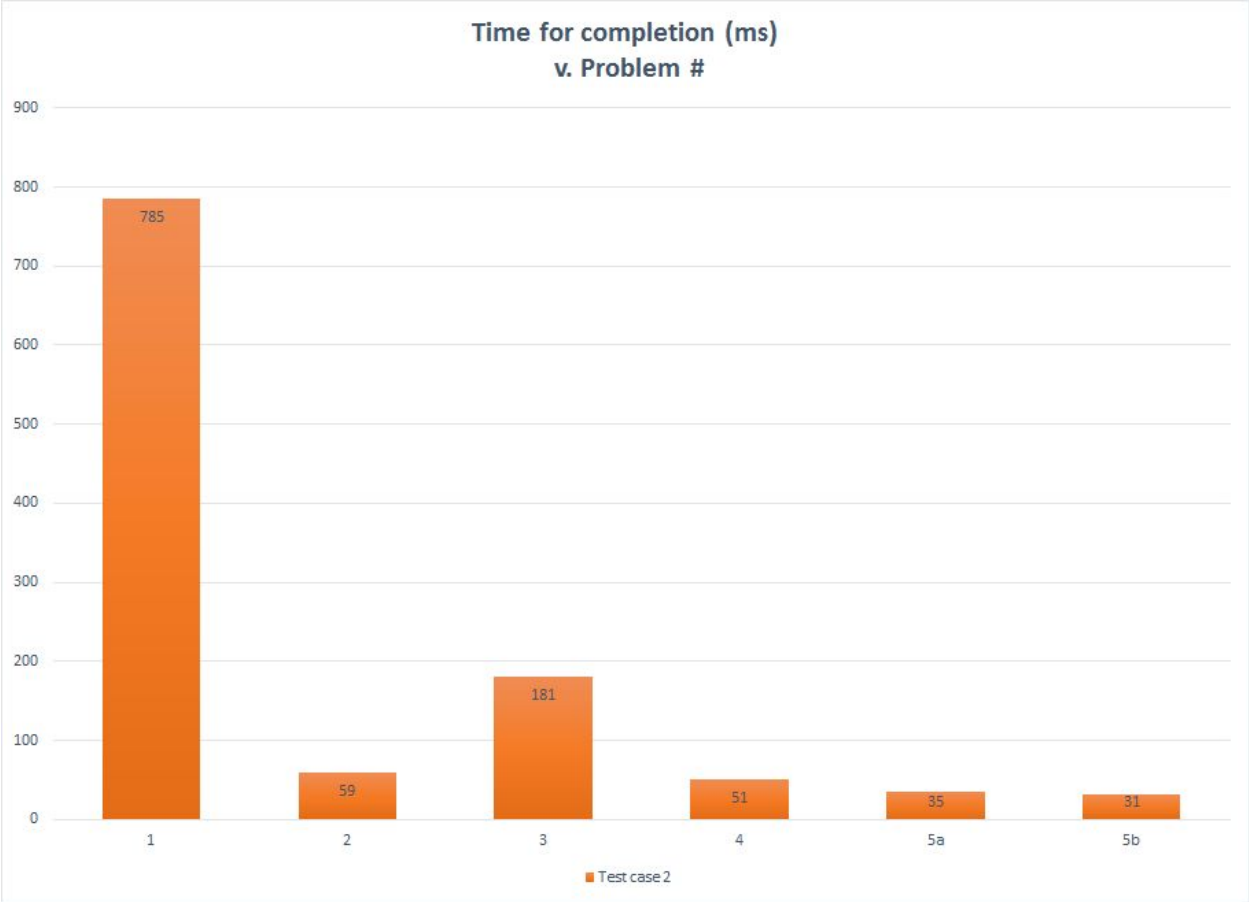
Graphs:

Test Case 1:



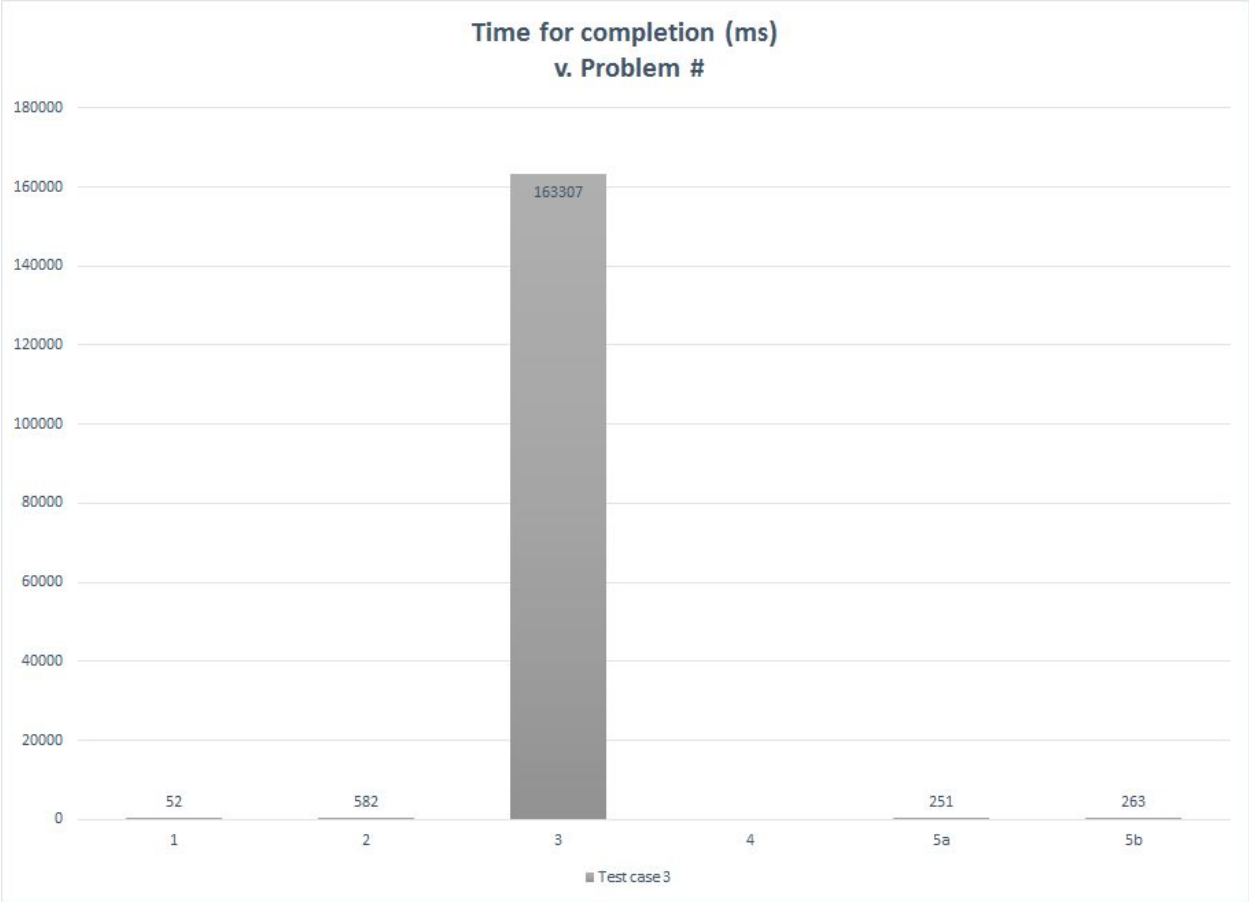
Test case 1 was a relatively small test case and thus the expected efficiencies are not apparent here. The blind search algorithms (1 and 2) take about twice as long as the others because they waste resources exploring unnecessary branches.

Test Case 2:



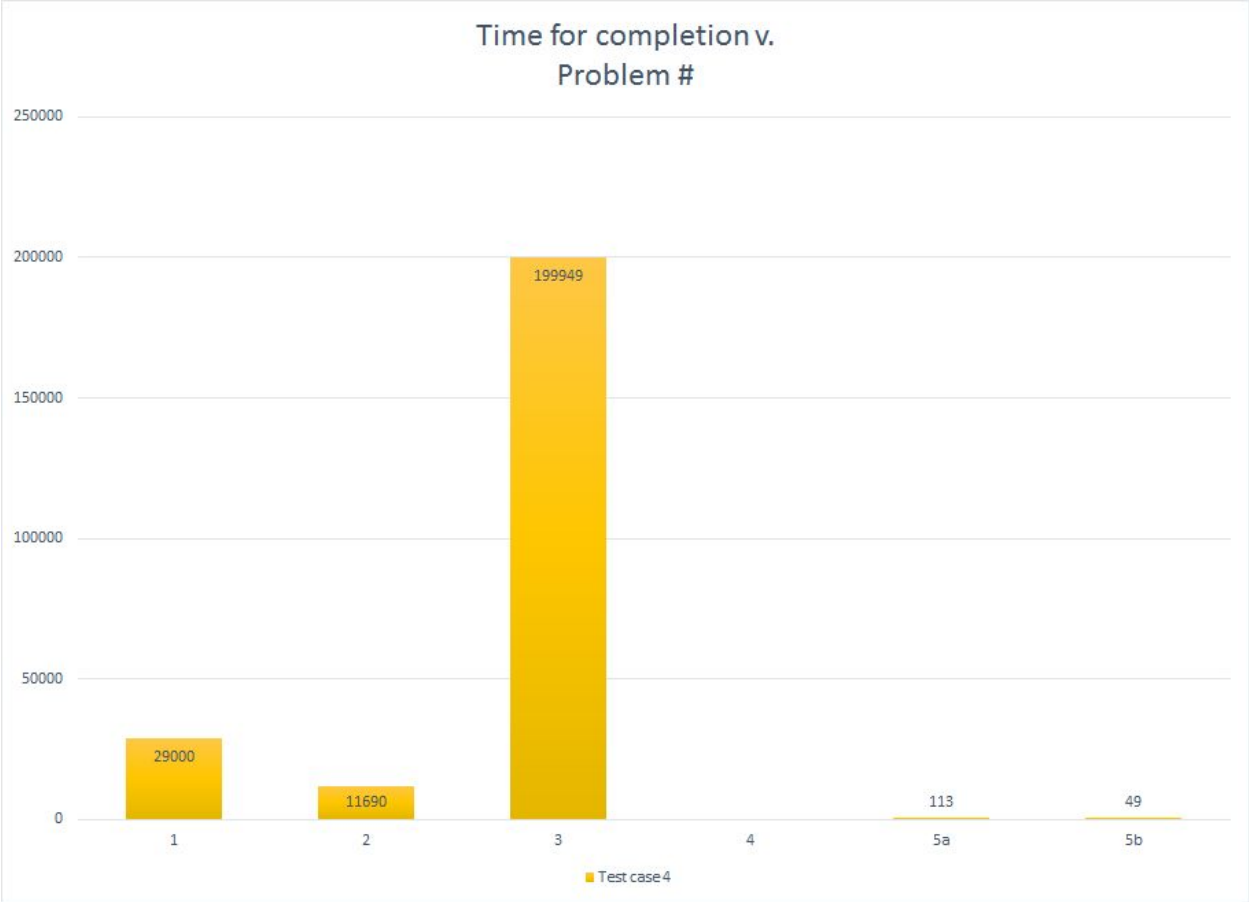
Test case 2 was much larger than the previous one. And here, the depth first search does quite well as compared to before because of the way the search tree is constructed and explored. The last three searches behave as expected (in being relatively fast).

Test Case 3:



Test case 3 compared two numbers of different length. The blind searches are expected to take a while since they must completely search the tree before concluding failure. The iterative deepening depth first search is expected to significantly longer because it searches parts of the tree multiple times (over all of its iterations) before it can conclude at the highest allowed depth that the path does not exist.

Test Case 4:



Test case 4 is the largest test case and looks closer to expectations. The heuristic search is the fastest because it prioritizes branches which look more appealing. The DLS beats the BFS because the the numbers we selected are close enough together to be discovered on one side of the tree. Again, the iterative deepening depth first search takes longer because it must go though all versions of trees of depths less than the optimal solution's depth before it can find the solution.

- A paragraph from each author stating what their contribution was in this assignment.

Authors:
Ketan Kelkar:

I have never used python in the past, so I was not able to contribute significantly to the actual code written. I worked with Soham to create the reachable configurations generator and primality testing functions and pseudo-code the breadth first search solver. I worked with Ajay to conceptually figure out how to apply the heuristic search. I wrote a small helper function which calculates the hamming distance between two integers. We worked as a team on the depth first search. We all discussed the content

of the report, and I compiled much of it. With the help of the others, it was brought to completion.

Soham:

I wrote the algorithms for Breadth First Search, Depth First Search, Iterative-Depth First Search, and Bi Directional Search. Developed the code and tested each code. Also, extensively debugged the code to get rid of any bugs found. I also worked on developing the Heuristic Cost function for the part B of problem 5. I also collaborated with both my team members to incorporate their ideas and tips to optimise the code.

Ajay Bhullar:

I wrote the algorithm for the A* search(using Ketan's hamming distance helper function) and slightly modified the helper function to account for the path cost as well. In addition to this I generally helped Soham and Ketan debug the other problems from a conceptual as well as coding standpoint. I am not well versed in python, as a result Soham was the primary writer of code. All of us were present during most of the programming sessions and provided feedback conceptually. I also assisted in testing/debugging all parts of assignment 1(except problem 4).