For each problem in the assignment, we attempted to convert a specified prime number, into different prime number. The conversion had to be done one step at a time changing a single digit at each step. At all times, the number had to be prime. If there was no possible way to reach the desired prime number from specified prime, then an UNSOLVABLE output was returned. Each problem was the same with a different algorithm used in order search for the solution.

For problem 1, the algorithm was a breadth first search, so every node of the highest depth is expanded before the nodes at the next depth are expanded. For problem 1, we used a queue and a dict. The queue was used to implement the first in first out nature of a breadth first search, while the dict was used to check if a node had been previously checked before.

For problem 2, the algorithm was a depth first search where the maximum depth checked was 5. For every node, we search its children to the maximum depth before checking the next node. To implement the depth first search, we used a list. The list used append and pop in order to have the last in first out functionality that a depth first search uses. This way, the most recent node put in will be the first node that is expanded.

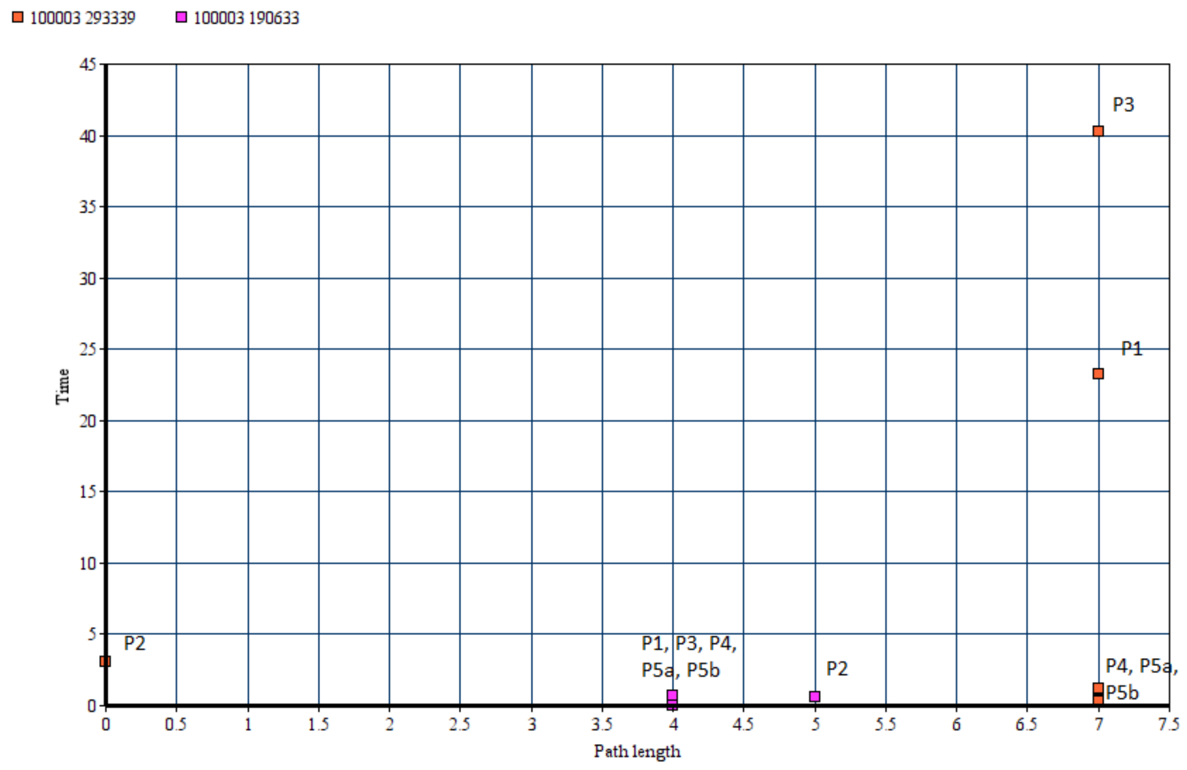
For problem 3, the algorithm was an iterative deepening depth first search where the absolute maximum depth was 8. We used the same idea as for problem 2, but began with a maximum depth of 1 and iteratively increased the maximum depth by 1 until a maximum of 8 or until a solution was found.

For problem 4, the algorithm was a bidirectional search. For this algorithm, we search forward from the starting prime and backward from the ending prime at the same time, and return if the lists reach a node that they have in common, or when one reaches the start/ending prime. Both the forward and backward searches performed breadth first searches in order to find a solution. For this we used 2 dicts and 2 queues which had the same function as with the breadth first search.

For problem 5 part a, the algorithm used was A*, where the heuristic was the hamming distance from the desired solution. To calculate the hamming distance, distance was increased by one for every digit that was not the same as the desired digit. In order to implement the A* algorithm, we used a priority queue and a dict. The dict was used to store already visited nodes and the priority queue was used to calculate which node to visit next. The next node was determined based on lowest cost, where cost was calculated using the hamming distance added with the distance already travelled.

For problem 5 part b, we use the same heuristic as part a, however when there is a tie in the priority queue, choose the value closest to the desired prime number.

Each of these algorithms handle the same input differently. Depending on the algorithm, the same input can result in a different answer, and take a different amount of time. Some may not find an answer at all if its constraints don't allow it to find a solution. With a simple input like 101 to 103 for instance, all the algorithms go straight from 101 to 103, except for depth first search. Depth first search ends up finding the solution 101 401 601 607 107 103 before it ever reaches the simplest solution of 101 103. Additionally, the depth first search takes 0.004 seconds, a significantly greater amount of time to reach its solution compared to the other answers. The breadth first search ends up taking 0.002 seconds, the 2nd highest amount of time while the other 3 algorithms are fairly close in time taking around 0.0006 seconds each. While the breadth first search takes more time in this simple case compared to the iterative depth first search, as the solution gets more complicated, the breadth first search ends giving much faster results. With a significantly more complicated input like 100001 293339, breadth first finds its solution in 40 seconds, while iterative deepening takes 4 minutes in order to reach a solution. Depth first however is unable to ever find a solution due to the depth being limited to 5 while the solution requires a depth of 7. Meanwhile, bidirectional and A* find an alternate solution that is also of depth 7, but only take a second in order to do so. A* finds the solution significantly faster because of its priority queue, which allows it to avoid expanding the large number of unnecessary nodes that breadth first search expands. The bidirectional search opens a lot of unnecessary nodes since it implements a breadth first search, however since it searches from both sides, it only search to a depth of 3 and 4 for each of its queues. As the depth increases, the nodes increase exponentially, so searching a depth of 3 and 4 is significantly faster than searching a single depth of 7.



As seen in the graph, as the path length increases, the time it takes to find a solution increases. The time increase is most noticeable for problems 1 and 3 where it increases exponentially as finding a path length of 4 took under a second, while searching a length of 7 took 24 and 40 seconds respectively. Problem 2 would likely face similar time issues if its maximum depth searched was increased, however since it only searches a depth of 5 at most, it can return UNSOLVABLE much faster. Problem 2 is also the only one to not return the shortest path in many cases due to it using depth first search.