__author__='jix029@ucsd.edu,A99076165,jih089@ucsd.edu,A99037641,z6hu@ucsd.edu,A99056145'

• Description of the problem and the algorithms used to solve problems 1 - 5.

1. simple breadth first search, on each while loop we expand the current prime number popped from the queue and added those adjacent prime numbers(primes that can be obtained by changing one digit on the current) and their path(parent's path + itself) in pairs back to the queue. When it runs, it will expand the prime numbers on the lowest level first then goes onto the next level until it finds the goal and print out the according path.

2. depth-limited search. We use recursion to expand the leftmost prime number that the current prime number can generate by changing one digit. Since the most deepest level is 5, the function should return "failure" if it could not find the final prime if level 5 has already been reached. Then backtracking to the upper level to explore another prime number that can be generated by current prime number. If the final prime has been found, return the final prime. If return value from the lower level is "failure", then continue searching, otherwise, return the string that concatenate the current prime number, space and the return value from the lower level.

3. iteratively deepening depth-first search. This is a depth first limited search with continuely increasing iteratively limited level until the limit of 8. We start depth first limited search with 0 level limit, and increase level limit by one if the final prime could not be found. Continue searching until the level limit reachs 8.

4. 2 BFS running simutanously. One from the starting prime, and the other from the final prime. When either queue becomes empty, it returns unsolvable because it seems that two paths have no way intersect in the middle. In the while loop, each BFS pops the prime from its queue, get the neighbors and put them into its dictionary, while checking if the neighbor already exists in the other dictionary. If it does, it finds the intersection and will return the path of this neighbor plus the path from the other dictionary using this intersection as the key. Otherwise, it goes to the next loop.

5. BFS using a priority queue. For part a, we calculate the total cost that is the sum of Hamming distance and the current path cost when we explore the neighbors of the current prime number and add it to the queue with neighbor. Then we always pick the next prime number which has the lowest total cost in the queue and expand on it. For part b, we choose a different heuristic which calculates the difference of the two numbers, we'll first explore the one with a smaller difference. It doesn't guarantee the optimal path.

• Describe the data structure used in each algorithm

1. we use a set to store all prime numbers that have been visited. A queue for BFS holding all the candidate prime numbers. We use a list to store all prime number that could be generated by changing one digit from the current number.
2. we use a set to store all prime number that have been visited on the current path from the root node to the current node.We use a list to store all prime number that could be generated by changing one digit from the current number.

3. we use a set to store all prime number that have been visited on the current path from the root node to the current node.We use a list to store all prime number that could be generated by changing one digit from the current number.
4. 2 queues storing unvisited prime numbers for 2 BFS. 2 sets holding visited prime number for 2 BFS. 2 dictionaries marking the path for each prime number.We use a list to store all prime number that could be generated by changing one digit from the current number.
5. a priority queue that is sorted according to the hamming distance. a set that stores visited prime number. We use a list to store all prime number that could be generated by changing one digit from the current number.

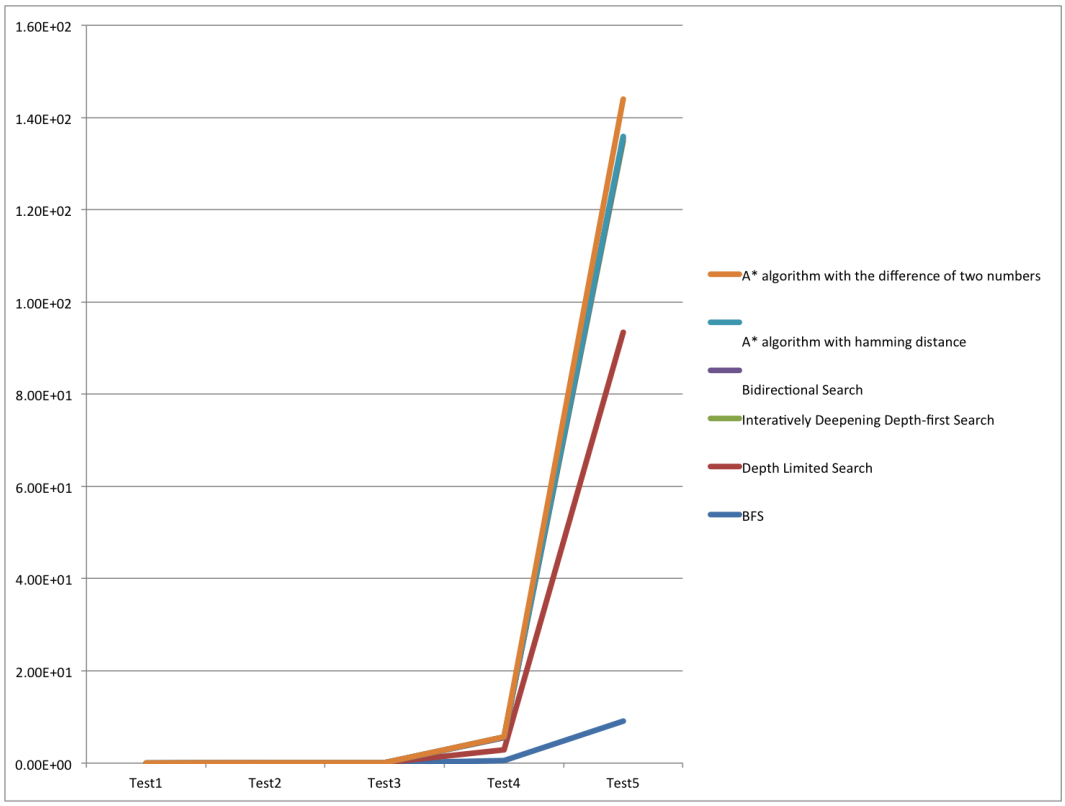
6.

Suppose **b** is branching factor, **d** is depth of solution, **m** is maximum depth, **l** is depth limit

Algorithms	Optimal?	Complete?	Time	Space	Test1: 2 5 Test2: 23 67 Test3: 103 199 Test4: 1327 2411 Test5: 10099 20011	Runtime (seconds)
BFS	Yes	Yes	$O(b^d)$	$O(b^d)$	Test1: 2 5 Test2: 23 13 17 67 Test3: 103 193 199 Test4: 1327 1427 1447 2447 2417 2411 Test5: 10099 10091 30091 30011 20011	Test1:0.000136 Test2:0.001207 Test3:0.002795 Test4:0.590348 Test5:9.132378
Depth Limited Search	No	Yes, if l >= d	$O(b^l)$	O(bl)	Test1: 2 1 3 5 Test2: 23 13 43 41 61 67 Test3: 103 503 523 593 193 199 Test4: 1327 1427 1447 2447 2417 2411 Test5: 10099 70099 70019 90019 90011 20011	Test1:0.000209 Test2:0.003637 Test3:0.025417 Test4:2.31221 Test5:84.33565
Iteratively Deepening Depth-first Search	Yes	Yes	$O(b^d)$	O(bd)	Test1: 2 5 Test2: 23 13 17 67 Test3: 103 193 199 Test4: 1327 1427 1447 2447 2417 2411 Test5: 10099 10091 30091 30011 20011	Test1: 0.00013 Test2: 0.00189 Test3:0.004563 Test4:2.597781 Test5:41.36279
Bidirectional Search	Yes	Yes	$O(b^{d/2})$	$O(b^{d/2})$	Test1:2 1 5 1 Test2:23 13 17 67 17 Test3:103 109 199 109 Test4:1327 1427 1447 2447 2411 2441 2447 Test5:10099 10091 30091 20011 30011 30091	Test1:0.000298 Test2:0.001491 Test3:0.002146 Test4:0.07596 Test5:0.664610
A* algorithm with hamming distance	Yes	Yes	Exponential	O(bm)	Test1: 2 5 Test2: 23 13 17 67 Test3: 103 193 199 Test4: 1327 1321 1381 2381 2311 2411 Test5: 10099 10091 30091 30011 20011	Test1:0.000252 Test2:0.001787 Test3:0.002719 Test4:0.047362 Test5:0.343441
A* algorithm with the difference of two numbers	No	Yes	Exponential	O(bm)	Test1: 2 5 Test2: 23 73 71 61 67 Test3: 103 193 199 Test4: 1327 1627 1697 1997 1999 1979 1879 2879 2579 2539 2339 23892381 2371 2377 2347 2447 2417 2411 Test5: 10099 17099 17093 17053 15053 1013 2013 21013 1011 20011	Test1:0.000249 Test2:0.001276 Test3:0.002235 Test4:0.064334 Test5:8.187881

Running time vs. Prime number: Based on the running time we get, all of the algorithm tend to increase exponentially, which matches what we expect from the Big-O notations. Generally, two A* algorithms take less time as the prime number gets larger because they avoid some redundant states. Breadth related algorithms (BFS and Bidirectional Search) perform better than Depth related algorithms in our test cases. It might because the depth solution in our test case is less than the depth limit, thus depth limited search involves some unnecessary steps. Meanwhile, interatively deepening depth-first search repeats some steps which takes longer. Bidirectional search takes less time than BFS because it generates from the start state and the goal state simutanously.

Graph:



Jinye Xu:
Worked on BFS related problems and debugging problem 2 and 3. Most of my involvement was on problem 1, 4 and 5.

Jiaying Hu: Mostly worked on DFS related problems and reports on these parts. Measure the running time for each algorithm.

Zhongting Hu: Modified the method of checking prime numbers and analyzed the relationship of all of the algorithms. Most of my involvement was on problem 6.