Joey Elsisi jre3wjh

Lab 108

December 5, 2019

Pre-Lab:

To begin my topological.cpp, I read in two strings, initialize my counter int, an unordered_map, and a vector. These are all big theta(1) constant time operations. They would also be big theta(1) constant space operations, with the int taking up 4 bytes, the empty unordered_map taking up 56 bytes and the empty vector taking 24 bytes. The space complexity of the two strings is dependent on the size of the inputs, with a big theta(# of chars) linear complexity, not the number of inputs. For this implementation the sizes will never exceed the uninitialized default size of 32 bytes and is therefore constant space complexity.

Now for reading in input, a while loop runs for the number of lines in the input .txt. This while loop is big theta(e) linear time, where e is the number of edges. Each time two strings are read in, a constant time .find() method within an if statement is called on the unordered_map to check if the string is a new node, these two if statements are big theta(1) constant time. I choose to us an unordered_map mostly for this reason. In the worst case that every line in the input file contains a new node, a constant time insertion into the unordered_map is done, and big theta(v) linear time push_back is done to my array. After the two if statements, I then update my adjacency list implementation. This requires a big theta(1) access of the unordered_map to specify the list to be traversed, than in the worst case that every node is adjacent to the node in question, a big theta(v) time is required to traverse all those edges. As for space complexity, a new node is created big theta(v) times. These nodes take 48 bytes of memory each. Both the unordered_list and the adjacency list store node pointers. They both will take up big theta(v) space. All in all reading in the input will take big theta(v^2) time.

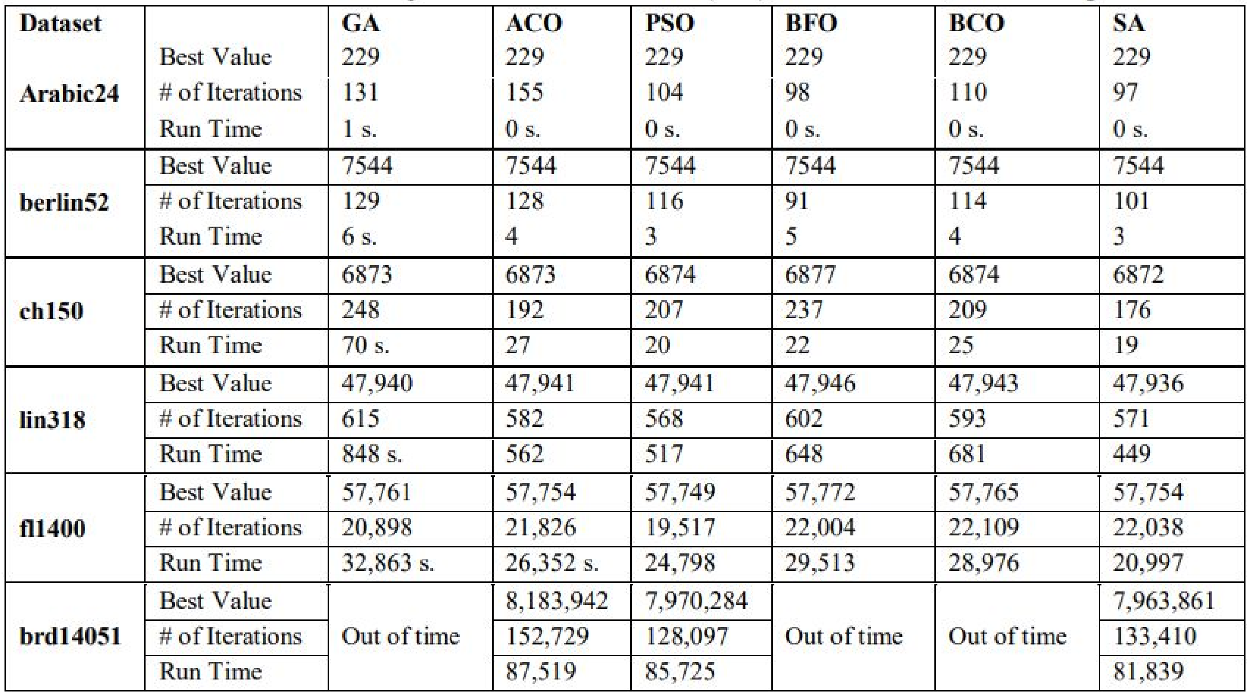
Now that reading the input, storing the unordered_map, and creating an adjacency list are all done, I call my topological sort method called topsort. First, an uninitialized queue is stored, taking 80 bytes. A 32 byte string and two 4 byte ints are also stored, all big theta constant space operations. A big theta(v) for loop is run to step through every node in my adjacency list. Within this for loop is a big theta(1) constant time access of the element at location i in the list, and a big theta(v) while loop that checks all the adjacencies of that node, which in the worst case would mean being adjacent to every node in the graph. I perform a separate while loop that pushes all the nodes that are to be deleted next. In the worst case that the queue has to be resized, this is a big theta(v) operation. As for space complexity, since I am passing pointers, for the entirety of my topsort method I do not have to create any new nodes. I perform one more while loop to create the return string. Within this loop I have another loop. These both in the worst case take big theta(v) time, but since my second while loop has a queue push, which is also big theta(v), These loops will take in total big theta(v^3) time. If I could do this lab over again, I would have initialized a proper size for the queue to prevent this. My total program time is big theta(v^3).

In-Lab:
I begin by reading in input, a big theta(1) constant time operation. I then initialize a middle earth object. With a push_back inside a for loop that iterates through all the possible cities, this constructor takes big theta(n^2) time. As for space complexity, the initialized MiddleEarth object takes 144 bytes, and is a constant size as both the x and y lengths of the map are stored as integers. getItinerary() is called, and a for loop is called to create a vector of cities. The vector starts at 24 bytes and will grow at big theta(n) for the number of cities available. The same situation as in the constructor is in this method, and will run in a big theta(n^2) runtime, where n is the number of cities chosen. Then I sort the vector of destinations, which according to this website, https://en.cppreference.com/w/cpp/algorithm/sort, is big theta(nlogn). I then have a while loop that runs for every possible permutation of the vector. Within this while loop is a big theta(n) linear compute distance method. The total time it takes to run through each permutation takes big theta(n!). As for the running time, the n! while loop dwarfs all other operations, making this program big theta(n!).

Acceleration:
I used this paper for my answers.
https://www.hindawi.com/journals/cin/2016/1712630/
https://pdfs.semanticscholar.org/080f/f5f7eea2cf18d13ab939fa5a2d7e0f12e234.pdf
Approximate solution: List based Simulated Annealing is an algorithm inspired by the controlled cooling of materials to reduce defects. This algorithm has several parameters you have to define on your own to use, making it pretty difficult to generalize this algorithm as some parameters work better for some problems than others. In each iteration, a random path is chosen. If that path is better, it is always switched to. If not, there is a small probability which you pick beforehand, called the temperature, that the less effective solution will be switched to either way. Every time a bad move is chosen, the temperature is reduced (called the cooling schedule), until at some chosen temperature level the algorithm is stopped. Unfortunately none of the papers written about this algorithm mention a run time and I can not understand the equations for the probability of getting a better path. But what is really interesting in small input sizes, the algorithm outperforms most as seen in this table. SA is the Simulated Annealing

algorithm.

Table I Results Of All Techniques To Measure Best Value (Path), No. Of Iteration, And Running Time



Dataset		GA	ACO	PSO	BFO	BCO	SA
Arabic24	Best Value	229	229	229	229	229	229
	# of Iterations	131	155	104	98	110	97
	Run Time	1 s.	0 s.	0 s.	0 s.	0 s.	0 s.
berlin52	Best Value	7544	7544	7544	7544	7544	7544
	# of Iterations	129	128	116	91	114	101
	Run Time	6 s.	4	3	5	4	3
ch150	Best Value	6873	6873	6874	6877	6874	6872
	# of Iterations	248	192	207	237	209	176
	Run Time	70 s.	27	20	22	25	19
lin318	Best Value	47,940	47,941	47,941	47,946	47,943	47,936
	# of Iterations	615	582	568	602	593	571
	Run Time	848 s.	562	517	648	681	449
fl1400	Best Value	57,761	57,754	57,749	57,772	57,765	57,754
	# of Iterations	20,898	21,826	19,517	22,004	22,109	22,038
	Run Time	32,863 s.	26,352 s.	24,798	29,513	28,976	20,997
brd14051	Best Value	Out of time	8,183,942	7,970,284	Out of time	Out of time	7,963,861
	# of Iterations		152,729	128,097			133,410
	Run Time		87,519	85,725			81,839

With enough time on my hands this would be my algorithm of choice.

Exact solution: The held-karp algorithm was derived in 1962. What this algorithm does is it breaks down the paths into smaller paths, computing their distances. This creates a sort of tree with subpaths already calculated. It recursively uses the smaller distances to solve greater distances. With shorter computed distances being stored, this algorithm uses up a ton of space, specifically big theta((2^n)(n)). In return, it computes the precise answer in big theta((2^n)(n^2)). I would definitely chose this algorithm to use after freeing up some space on my computer. This article helped me to understand this concept.
https://medium.com/basecs/speeding-up-the-traveling-salesman-using-dynamic-programming-b76d7552e8dd

Third solution: Branch and bound is an algorithm that uses minimum and maximum bounds to compute the shortest path. The first step is to use your adjacency matrix to find the minimum and maximum possible distance, or the initial lower bound and upper bound to choose your next vertex. Then, you create a search space tree from your initial node. Iterating through this tree will get you to the shortest path, which is done by going down level by level, instead of going through the entirety of a path on the tree. If a node in the level happens to be below the minimum distance or above the maximum distance, that path can be discarded. In the worst case, this algorithm is still theta(n!), but in practice it works really well. If I had to redo my lab, this would be the most practical choice, as it is fairly easy to implement. This youtube video explained the concept to me. https://www.youtube.com/watch?v=1FEP_sNb62k