Project 4: TSP

Bryant Hayes, Keenan Bishop, Nathan Ahrens

Project Report
You will submit a project report containing the following:

- the ideas behind your algorithm a completely as possible
- your “best” tours for the three example instances and the time it took to obtain these tours.
- Your best tours for the competition test instance(s).

OUR ALGORITHMS

In order to calculate tours for the travelling salesman that are optimal we found we needed to just try every possible combination of tours and keep track of the best one. This is very slow and is why the TSP problem is considered NP-Complete. Given that it is not efficient to find the optimal tour we have used a few algorithms designed to help us find a tour that is ‘close’ to optimal in much less time. To do this we thought about using a variety of algorithms:

Greedy
2-opt
3-opt
k-opt
Genetic
Simulated Annealing
Neural Network

From these we chose to investigate the ‘Greedy’ and the ‘2-opt’ algorithms because they seemed feasible in the time available and they seem like they offer promising solutions in minimal time. We started with the Greedy algorithm.

The concept of the Greedy Algorithm for solving the TSP is simple.Wherever you are, go to the nearest vertex that you haven’t been to before. For the test text file we were given then we merely read in the data and created a number of ‘cities’ which for us are just structs/objects that contain an id, x and y variable.From the text file’s data we were able to assign all three of these object’s variables. With the cities available in memory we could merely start at index 0 and use a for loop to iterate through every other vertex and calculate the distance from the current vertex. Once we found the smallest distance we would remove the initial city from the ‘remaining_cities’ list and travel to that nearest city. We continue this pattern until there are no more cities on the ‘remaining_cities’ list and then add one more connection back to the starting city. This gets us a result that is correct, but looking at the generated plot it is obvious that it is not that efficient.

In order to make a more optimal tour we use the 2-opt algorithm. This algorithm requires that you have an existing tour, so we use the solution from our greedy approach we talked about previously. The 2-opt algorithm works by making slight changes and seeing if that change helped. More specifically it swaps two edges so that the path is still connected and sees if that swap made the overall distance shorter. It repeats this until the swapping of any two edges results in no further improvement. We found that this algorithm was not that difficult to conceptualize, thus not too hard to program in python. Once we had finished however, we released that the high end programming language of python was not the best for crunching large algorithms with a lot of looping. When run on some of the test cases the greedy algorithm would finish quickly but then the program would take quite a lot of time to finish the 2-opt algorithm. Due to this, and the fact that we want to do well in the competition we decided to re-write the program in C, which we knew would be faster as there is less abstraction and higher level programming overhead. This took longer than anticipated and showed us how much quicker it is to write code in python than in C.

The 2-opt algorithm is pretty fast compared to some of the other algorithms, but because it uses a lot of ‘trial and check’ it still is not very fast for large numbers of cities. Below is an excerpt of the python code which performs the edge swap as well as the code that checks to see if the swap made an improvement to the overall distance.

```
def run2opt(tour, currentDistance):
    for i in xrange(len(tour)-1):
        for k in xrange(i + 1, len(tour)):
            newTour = optswap(tour, i, k)
            newDistance = tourDistanceSum(newTour)
            if newDistance < currentDistance:
                return(newTour, newDistance)
    return(tour, currentDistance)

def optswap(tour, i, k):
    newTour = []
    for j in xrange(0, i):
        newTour.append(tour[j])
    for j in xrange(k, i-1, -1):
        newTour.append(tour[j])
    for j in xrange(k + 1, len(tour)):
        newTour.append(tour[j])
    return newTour
```

RESULTS

The two main tests are are concerned with are the test case text files:

tsp_example_1.txt
tsp_example_2.txt
tsp_example_3.txt

and the competition test case files:

test-input-1.txt
test-input-2.txt
test-input-3.txt
test-input-4.txt
test-input-5.txt
test-input-6.txt

For the main test case files (non-competition) the following table shows our shortest distances as well as the time it took to achieve that tour. The table shows this data for the C version. Below that table is the table for the competition results. **This data shows timing and distance for the basic greedy followed by 2-opt algorithm.**

File	time © -seconds-	best distance (C)	verified with tsp-verifier.py?
tsp_example_1.txt	.009	110722	Y
tsp_example_2.txt	.967	2785	Y
tsp_example_3.txt	390	1964254	Y

File	time (C) -seconds-	best distance (C)	verified with tsp-verifier.py?
test-input-1.txt	.001	5639	Y
test-input-2.txt	.016	7669	Y
test-input-3.txt	1.12	12637	Y
test-input-4.txt	17.5	17784	Y
test-input-5.txt	390	28685	Y
test-input-6.txt	390	63611	Y
test-input-2000.txt	390	39629	Y

ISSUES / IMPROVEMENTS

There are definitely things we could still optimize in the C code to make it even faster. Using a profiling tool we were able to see which parts of our code were taking the most time.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 48.01      2.30     2.30 504280875     0.00     0.00  distance
 24.63      3.49     1.18  2016997     0.00     0.00  opt_swap
 22.55      4.57     1.08  2016999     0.00     0.00  sum_tour_distance
  4.59      4.79     0.22                             round
  0.42      4.81     0.02      126     0.00     0.04  run_2_opt
  0.00      4.81     0.00      250     0.00     0.00  remove_city
  0.00      4.81     0.00        1     0.00     0.00  build_cities
  0.00      4.81     0.00        1     0.00     0.00  greedy_algorithm
  0.00      4.81     0.00        1     0.00     4.59  opt
  0.00      4.81     0.00        1     0.00     0.00  read_file
```

As you can see above, the ‘distance()’ function is taking a significant amount of time merely because of the frequency of which it is called. If we could eliminate the need to recalculate the distance each time we might be able to speed up the program. One idea we had was to store the distance between each combination of cities in a large dictionary, but that would sacrifice a large amount of space for only a possible increase in speed, or so we thought. When we actually implemented the system which gave each city an array containing all the distances to other cities we found that the number of calls to ‘distance()’ went from ~504,000,000 down to ~62,500. **This made the program execute 15 times faster!** This was a major improvement to our code and made us able to finish the 2-opt algorithm for 1000 cities in under the 5 minute limit.

We had to allocate some time to fixing a memory issue where millions of bytes were getting malloc’d without a purpose and once we moved around some variables and free’d some memory after we were done with it, it seemed to fix the errors’ we were getting. We took advantage of the tool ‘valgrind’ to help us there.

CONCLUSION

We believe our algorithms are fairly complex enough that they get close to optimal results in a relatively short amount of time. For more information about running, compiling and using our code please refer to the README file in the project4 folder.

In regards to our actual tours, all of our solutions have been been calculated and saved into the <filename>.tour files in the project4/completed_tours folder. For example our solution to: *tsp_example_1.txt* can be found in *tsp_example_1.txt.tour*.