Kelvin Lu (lukelv)
Alvin Le (lealv)
Amanda Grant (granaman)
CS325 Winter 2018
Final Project TSP Report

Project Report

You will submit a project report containing the following:

- A description of at least three different methods/algorithms for solving the Traveling Salesman Problem along with pseudocode. Summarize any other research your group did.
- A verbal description of your algorithm(s) as completely as possible. You may select more than one algorithm to implement.
- A discussion on why you selected the algorithm(s).
- Pseudo code
- Your "best" tours for the three example instances and the time it took to obtain these tours. No time limit.
- Your best solutions for the competition test instances. Time limit 3 minutes and unlimited time.

Methods and algorithm research

Group discussion thread:
https://oregonstate.instructure.com/groups/295367/discussion_topics/8169760

Brute Force

Prepared by Amanda Grant

Here's an entertaining read on the brute force method:
https://medium.com/basecs/speeding-up-the-traveling-salesman-using-dynamic-programming-b76d7552e8dd

Brute force is very slow and we shouldn't consider it for our project. Nearly all alternatives are more performant. That's because brute force runs in O(n!).

If the salesman has to visit 6 cities in any order (and return to his starting city at the end) that's 5! possible circuits to consider. 5 x 4 x 3 x 2 x 1 = **120**.

On the bright side, this method is guaranteed to find a solution.

But it quickly becomes infeasible, even for seemingly small quantities of cities. Let's say we have just 15 cities. That's 14! possibilities to calculate and compare to find the shortest path.

14! = 87,178,291,200

87 billion possibilities to compare is crazy, and would take forever to calculate. Even if we could calculate a million possibilities per minute, we would need 1,000 minutes to calculate a billion possibilities (16 hours) and 1,392 hours (or so) to get 87 billion possibilities. That's 58 days - no one's going to sit around for that.

So brute force is definitely out.

Ant colony

Prepared by Amanda Grant

The "ant colony" approach to solving the traveling salesman problem was inspired by the observation that real life ants are able to find the shortest path from their colony to a food source.

Sources

- https://www.sciencedirect.com/science/article/pii/S1002007108002736 This source is math and formula heavy. I don't possess the math background to parse some of what's in here but it has a well-written explanation of how the ant colony method works.
- This video has a lot of filler but the important part is very short. Watch from about 0:45 to 1:40. Link to video

Overview

Every time an ant makes a complete trip between the colony and a food source nearby, it leaves a pheromone coating on the route it took. The more pheromone a route has, the more likely other ants are to pick it.

Ants that find a shorter route are able to make more trips in a time period, and since they are making more trips, they are coating the route with more coatings of pheromone, which means more ants will take that route in the future. Imagine the colony just continuously generates ants.

(The YouTube video has a nice animation demonstrating this perhaps better than words do.)

Eventually the best route has been identified and all the ants are using it.

This is easy to visualize if you have just one colony and one end node, with two different-weight paths to get there.

Our TSP challenge adds a few layers of complexity:

- we know the lengths of each edge upfront
- our food source is a series of nodes
- our ants must not revisit an already-visited nodes

Pseudocode

Main loop - it's based on "ticks" or "iteration numbers", represented by t, rather than the passage of real time. (You might recall something similar from Langton's Ant in CS162.)

```
for t = 1 to iteration number do
 for k = 1 to l do
   Repeat until ant k has completed a tour
     Select the city j to be visited next
     //For j, use probability given by equation 1
   Calculate Lk
 Update trail levels according to equations 2-4
```

(Pseudocode taken from the sciencedirect article)

There are several additional equations needed for calculating the probability of picking a city and the updating of the pheromone levels on each trail. This is where my understanding of this problem completely breaks down, I don't know how to interpret these equations.

Equation 1 - probability of picking a particular city

$$p_{ij}^k = \begin{cases} \frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{s \in allowed_k} [\tau_{is}]^\alpha \cdot [\eta_{is}]^\beta} & j \in allowed_k \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

Equations 2-4, used in support of equation 1

$$\tau_{ij}(t+1) = \rho \cdot \tau_{ij}(t) + \Delta\tau_{ij} \quad (2)$$

$$\Delta\tau_{ij} = \sum_{k=1}^{l} \Delta\tau_{ij}^k \quad (3)$$

$$\Delta\tau_{ij}^k = \begin{cases} Q/L_k & \text{if ant } k \text{ travels on edge } (i,j) \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

I am not confident in my ability to translate these formulas into pseudocode. I don't know what they do.

Should we attempt this algorithm?

It won't necessarily find the most optimal solution, but it **might be more accurate** than other approaches. (The charts under 5. Numerical simulation depict multiple runs of the algorithm and what the optimal path was vs. how close they came to it.)

Personally, I think the math on this one is complex. Maybe someone else in the group can break the formulas down into something we can use? It's beyond my abilities.

I think the idea of adding "weights" to already-explored, known-to-be-short paths has promise. Maybe we can take this idea into whatever we do end up implementing. Let's say we end up with an idea that finds as many possible routes as it can in the time given and then picks the shortest one. Adding a "preference" or "weight" to a particular sub-path might help future route discoveries come up with better complete paths than they would just guessing. Like if we know A-C-G is a very short path, maybe future runs should start with A-C-G.

TL;DR: I don't understand the math for "Ant Colony" well enough to suggest we implement it as-is.

2-opt Algorithm

Prepared by Alvin Le

Overview

This is an optimization algorithm that is applied to an already generated solution to the TSP problem. The algorithm essentially continuously swaps edges in the solution until the solution no longer improves via this swapping mechanism.

A swap involves removing two edges in the solution, thus leaving two separate paths. Then two different edges are added to reconnect these two paths. If the new graph has a better solution than the old one, then it is kept. If not, the change is discarded. Every possible valid combination of the swapping mechanism is performed.

All of this is just one iteration of the algorithm. We start again with a new solution, usually better than the previous, and perform every combination of the swapping mechanism again. We continue iterating this algorithm until either we cannot get an improved solution, or we can stop it after an arbitrary number of iterations.

The pre-generated solution can be developed from a good algorithm if time permits. Or we could quickly generate a poorly optimized solution, and then apply this 2-opt algorithm. An algorithm to generate a quick, but poor solution would be the greedy algorithm. We start at the first node and pick the closest node. Then from that node, we pick the next closest of the remaining nodes, and so on.

Pseudocode (obtained from the 2-opt Wiki page): https://en.wikipedia.org/wiki/2-opt

```
repeat until no improvement is made {
  start_again:
    best_distance = calculateTotalDistance(existing_route)
      for (i = 1; i < num of nodes eligible for swap - 1; i++) {
        for (k = i + 1; k < num of nodes eligible for swap;
k++){
          new_route = 2optSwap(existing_route, i, k)
          new_distance = calculateTotalDistance(new_route)
            if (new_distance < best_distance) {
              existing_route = new_route
              goto start_again
            }
          }
      }
```

```
   }
2optSwap(route, i, k) {
   1. take route[0] to route[i-1] and add them in order to
new_route
   2. take route[i] to route[k] and add them in reverse order to
new_route
   3. take route[k+1] to end and add them in order to new_route

   return new_route;
}
```

Example

I thought pseudocode for the algorithm, particularly the swapping part, was quite confusing. Even though we are swapping edges, the algorithm never actually works with edges. We can essentially swap edges by changing the ordering of the vertices. This is actually good for us, since our solution should look like a list of vertices taken in order.

I found it helpful to work through a very simple example. Consider the graph(excuse how terrible this looks, I couldn't figure out how to embed an image from my computer)

A---**1**--------B

| **2**\ / |

1 \/ **1**

| /\ |

| **3**/ \ |

C----**1**-------D

AB = 1, BD=1, CD=1, AC=1, AD=2, BC=3.

Lets say our initial solution is ABCDA, which has length 7. The optimal path is ABDCA or ACDBA for length of 4.

numNodes = 4, so i will loop from 1 to 2 and k will loop from 2 to 3.

i = 1, k = 2: The result from swapping is ACBDA. Same length as our current solution, so current solution ABCDA is kept.

i = 1, k = 3: The result from swapping is ADCBA. Same length as our current solution, so current solution ABCDA is kept.

i = 2, k = 3: The result from swapping is ABDCA, which is a better solution, so we update solution.

That ends the first iteration of the algorithm.

Since we got an improvement, we will repeat this all over again. But this time, we will not run into any better solution, so we stop.

Runtime

The outer for-loop runs from i = 0 to i < numNodes – 1. The inner for-loop runs from k = 1 to i < numNodes. Within the inner for-loop is the swap function, which is O(n). Thus, the entire algorithm is $O(n^3)$.

Should we use this?

Since this is an optimization algorithm that is applied to a solution, we could always run it after some other solution-generating algorithm if time permits. The non-competition part is not time-limited, so it could be safe to run this 2-opt algorithm on top of some other primary algorithm to get an improved solution.

Sources

https://en.wikipedia.org/wiki/2-opt (Links to an external site.)Links to an external site. - The Wiki article. Has good general information and pseudocode. A good description of the swapping mechanism.

http://cs.indstate.edu/~zeeshan/aman.pdf (Links to an external site.)Links to an external site. - Has illustration of steps in the 2-opt algorithm.

http://www.technical-recipes.com/2012/applying-c-implementations-of-2-opt-to-travelling-salesman-problems/ (Links to an external site.)Links to an external site. - example C++ implementation of 2-opt.

https://www.slideshare.net/kaalnath/comparison-of-tsp-algorithms (Links to an external site.)Links to an external site. - This one has comparisons of some TSP algorithms.

Held-Karp Algorithm

Prepared by Kelvin Lu

The Held-Karp algorithm is a dynamic programming approach for solving the TSP problem that can help reduce the number of function calls that a brute force solution would produce.

The algorithm takes advantage of the optimization property: “Every subpath of a path of minimum distance is itself of minimum distance” to generate subproblems from shorter paths, leading to an optimal solution.

https://www.youtube.com/watch?v=-JjA4BLQyqE

This YouTube video provides a great explanation for the methodology behind the algorithm. It is very similar to some of the other dynamic programming problems we have solved earlier in the class, especially the 0-1 Knapsack problem.

Pseudocode (Refined from Wikipedia to output both cost and the tour):

n = number of cities

Assume city ID’s are 1-n. Distance matrix d contains all distances d_{ij} where i is the first city’s ID and j is the second city’s ID

```
for k = 2 to n:
  Cost({k}, k) = d1,k
  Prev({k}, k) = 1
    for setLength = 2 to n-1:
      for all S where S = subset of {2,…,n} and |S| = setLength
        for all city c in S:
          Cost(S,c) = minm≠c,m∈S [Cost(S-{c}, m) + dm, c]
          Prev(S,c) = m corresponding to previous statement min
          totalCost = mincity c ≠ 1[Cost({2,…,n}, c) + d1,c]
          p = c corresponding to previous statement
          add cityID 1 to tour
          Set s = {2,…n}
          while p ≠ 1:
            add cityID p to tour
            S = S-{p}
            p = Prev(S,p)
       return tour and totalCost
```

Complexity:

Worst-Case time complexity: $O(2^n n^2)$

Space: $O(2^n n)$

Pros:

Exponential time vs. factorial time of brute force algorithm

Cons:

Exponential space requirements

Conclusion:

While better time complexity, exponential space requirements provide a separate massive overhead. I would only consider this option if there was not another choice for limiting the space requirements. However, the implementation seems relatively easy compared to the more complicated algorithms, except for maybe one issue that I foresee could occur is representation of the sets. It is also not an approximation and should always lead to the correct answer.

References:

https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm

https://www.youtube.com/watch?v=-JjA4BLQyqE

https://medium.com/basecs/speeding-up-the-traveling-salesman-using-dynamic-programming-b76d7552e8dd

Nearest Neighbor

Prepared by Amanda Grant

Nearest works a lot like you'd expect a greedy algorithm to.

Steps

1. Select a random city to start at
2. Go to the nearest unvisited city
3. Are you done? If not, go back to 2.
4. Return to the first city (done)

Pseudocode

```
tspPath = []
tripDistance = 0
unvisitedCities = list of cities left to visit
city = random city from the list of cities to visit
startingCity = city #keep a record of where we started for use
at the end
while (length(allCities) > 0)
  #get the closest city to the current city
  #method returns the closest city and its distance
  closest, distance = closestCity(city, unvisitedCities)
  tspPath.append(closest)
  unvisitedCities.remove(closest)
  tripDistance += distance

#now go back to the starting city from this current city
closest = startingCity
distance = distance(startingCity, city)
path.append(closest)
tripDistance += distance

#needs a distance helper method
#needs a closestPoint helper method that determines which other
city (from a list of possible cities) is closest to the given
city
```

Repeat for every possible starting city and then pick whichever starting city yielded the best (shortest) cycle.

Performance

It doesn't yield perfect results or the best performance, but it's regarded as a simple "good enough" choice. There's also a chance that it won't find a path when one does exist (I think this is just in the case that some cities can only be accessed from certain other cities, though? In our project, we can assume all cities connect to all other cities as per the group project discussion Q&A.)

Nearest Neighbor runs in $O(n^2)$

Should we use it?

Maybe - it's **easy to implement**, gets **decent results**, and it **might serve as a good first-step algorithm to then refine with 2-Opt** (see Alvin's post on 2-Opt).

Nearest Neighbor keeps its tours within 25% of the Held-Karp lower bound, which is a way of comparing the accuracy of various TSP algorithms. (Source: https://web.tuke.sk/fei-cit/butka/hop/htsp.pdf

More reading on Nearest Neighbor: https://link.springer.com/chapter/10.1007/978-3-319-00951-3_11

And a helpful video: Traveling Salesman Problem (Nearest Neighbor Algorithm)

In researching Nearest Neighbor, I found an improvement on it, which is just called Greedy Algorithm.

Greedy Algorithm

Prepared by Amanda Grant

This approach first **sorts all of the paths between cities** and then **repeatedly picks the shortest edge** as long as the edge chosen 1. hasn't been picked before, 2. doesn't create a cycle with less than N edges, or 3. doesn't increase the degree of any node to more than 2.

Pseudocode(ish)

```
1. Sort all edges
2. Select the shortest edge and add it to our tour if it:
  a. hasn't been picked before
  b. doesn't create a cycle with less than N edges (where N is
the number of cities to visit)
  c. doesn't increase the degree of any node to more than 2 (to
prevent re-visiting cities)
```

The Greedy algorithm normally keeps within 15- 20% of the Held-Karp lower bound (source: https://web.tuke.sk/fei-cit/butka/hop/htsp.pdf)

Should we choose this algorithm?

Maybe - it's slightly more complicated to implement than Nearest Neighbor, but it gets better results right out the gate. Then, we could refine the result with something like 2-Opt.

Christofides Algorithm

Prepared by Kelvin Lu

The Christofides algorithm is another approximation solution for the travelling salesman problem. This algorithm is specialized for distances that form a metric space – perfect for our specific problem where the distances are measured using the Euclidean distance formula. The algorithm guarantees that a solution will be found that is within a factor of 3/2 of the optimal solution. The Wikipedia page for this algorithm states that is the "best approximation ratio that has been proven for the traveling salesman problem on general metric spaces"

Pseudocode (Refined from Wikipedia):

1. Create a minimum spanning tree *T* of *G* (We can do this using Prim/Kruskal's algorithm).
2. Let *O* be the set of vertices with odd degree in *T*. By the handshaking lemma, *O* has an even number of vertices.
3. Find a minimum-weight perfect matching *M* in the induced subgraph given by the vertices from *O*.
    1. Perfect Matching:
    2. Given a graph $G = (V,E)$, a **matching** *M* in *G* is a set of pairwise non-adjacent edges; that is, no two edges share a common vertex.
    3. Perfect matching is a matching which matches all vertices of the graph. Every vertex of the graph is incident to exactly one edge of the matching
4. Combine the edges of *M* and *T* to form a connected multigraph *H* in which each vertex has even degree.
5. Form an Eulerian circuit in *H*.
6. Make the circuit found in previous step into a Hamiltonian circuit by skipping repeated vertices (*shortcutting*).

Example:

Wikipedia has a visual representation of the algorithm:
https://en.wikipedia.org/wiki/Christofides_algorithm

Another Representation in this lecture:
http://www.cs.princeton.edu/~wayne/cs423/lectures/approx-alg-4up.pdf

Analysis:

I really like this algorithm. Time complexity wise it should be very fast- I believe forming the minimum spanning tree may be the highest step cost [O(E lg V)] in the code if we use a greedy algorithm for the perfect matching. The approximation is also very accurate and can probably be improved even further with the 2-opt algorithm. Coding-wise I also think this should not be too hard to implement, as it mainly uses concepts that we have already learned before.

References:

https://en.wikipedia.org/wiki/Christofides_algorithm

http://www.cs.princeton.edu/~wayne/cs423/lectures/approx-alg-4up.pdf

http://www.cs.cornell.edu/courses/cs681/2007fa/Handouts/christofides.pdf

Verbal description of the algorithm(s) we chose

Christofides Algorithm

For data sets of all sizes we first use the Christofides algorithm.

This is a heuristic algorithm that first finds a minimum spanning tree of a given graph. Then, it looks just at the set of vertices with odd degrees. Using those vertices, it finds a minimum-weight “perfect matching”, which is the set of edges without common vertices. It then combines the edges from the minimum spanning tree and the edges of the perfect matching to create a connected multigraph. In this multigraph, every vertex has an even degree. Then, it finds an Euler circuit in that multigraph. Finally, it uses that circuit to find a Hamiltonian circuit by skipping any repeated vertices.

2-opt Algorithm

For smaller data sets (under 1000), we use the 2-Opt algorithm to improve our solution.

This is an optimization algorithm that is applied to an already generated solution to the TSP problem. The algorithm essentially continuously swaps edges in the solution until the solution no longer improves via this swapping mechanism.

A swap involves removing two edges in the solution, thus leaving two separate paths. Then two different edges are added to reconnect these two paths. If the new graph has a better solution than the old one, then it is kept. If not, the change is discarded. Every possible valid combination of the swapping mechanism is performed.

All of this is just one iteration of the algorithm. We start again with a new solution, usually better than the previous, and perform every combination of the swapping mechanism again. We continue iterating this algorithm until either we cannot get an improved solution, or we can stop it after an arbitrary number of iterations. The runtime is $O(n^3)$.

Why we choose the algorithm(s) we did

We chose to implement Christofides because it guarantees a solution within a factor of 3/2 of the optimal solution, which we also know to be the best approximation ratio that has been proven (so far). It looked reasonably doable with the time and knowledge we already have, too, using techniques and data structures we've learned in this course. We used Christofides to generate a solution and then refine it with 2-opt.

We chose to implement the 2-opt algorithm because it could improve any solution to the TSP problem. Since we had an unlimited amount of time for the primary grading, running this optimization algorithm could only help us. Also, the solution generating algorithm we chose, the Christofides algorithm, runs fairly quickly. Thus, we expected to have extra time to run the 2-opt algorithm for the competition instances.

Pseudocode of our algorithm(s)

Christofides

```
//Create distance matrix from cities
matrix = distMatrix(cities);

//Create MST adjacency list using Prim's
adjacencyList = primMST(matrix);

//Match odd vertices in the minimum spanning tree
//add those edges to the MST
match(adjacencyList, matrix);

//Find a euler cycle in the resulting adjacency list
euler = eulerTour(adjacencyList);

//Convert euler cycle to hamiltonian cycle
hamiltonian = hamiltonianTour(euler);

//Build the TSP tour from the hamiltonian cycle
tspTour = [];
for (int i = 0; i < hamiltonian.size() - 1; i++)
{
     cost += matrix[hamiltonian[i]][hamiltonian[i + 1]];
     tspTour.push_back(cities[hamiltonian[i]]);
}

//Add last vertex and edge to end cycle
int last = hamiltonian.back();
tspTour.push_back(cities[last]);
cost += matrix[hamiltonian[0]][last];

return tspTour;

//helper method for calculating distance between cities
distMatrix(cities)
{
      size = cities.size();
```

```
	matrix(size);
	for (int i = 0; i < size; i++)
	{
		for (int j = 0; j < size; j++)
		{
			matrix[i].push_back(distance(cities[i], cities[j]));
		}
	}

	return matrix;
}
```

2-opt Algorithm

```
repeat until no improvement is made {
   start_again:
   best_distance = calculateTotalDistance(existing_route)
   for (i = 1; i < num of nodes eligible for swap - 1; i++) {
      for (k = i + 1; k < num of nodes eligible for swap; k++){
         new_route = 2optSwap(existing_route, i, k)
         new_distance = calculateTotalDistance(new_route)
         if (new_distance < best_distance) {
            existing_route = new_route
            goto start_again
         }
      }
   }
}

2optSwap(route, i, k) {
   1. take route[0] to route[i-1] and add them in order to new_route
   2. take route[i] to route[k] and add them in reverse order to new_route
   3. take route[k+1] to end and add them in order to new_route
   return new_route;
}
```

Best tours for three example instances

Your “best” tours for the three example instances and the time it took to obtain these tours. No time limit.

Data set	**Best tour**	**Time to obtain**
tsp_example_1.txt	112560	0.09 seconds
tsp_example_2.txt	2791	3.03 seconds
tsp_example_3.txt	1885345	37.05 seconds

Competition solutions

Your best solutions for the competition test instances. Time limit 3 minutes and unlimited time.

Competition data set	Best tour	Time to obtain
test-input-1.txt	5803	0.043 seconds
test-input-2.txt	7799	0.381 seconds
test-input-3.txt	12889	7.367 seconds
test-input-4.txt	17944	43.435 seconds
test-input-5.txt	27648	0.199 seconds
test-input-6.txt	39345	0.811 seconds
test-input-7.txt	61331	5.160 seconds