CS325 Project Report - Group 56

Hunter Land
Tanner Rousseau
Cody Ray

Studied Algorithms

Nearest Neighbor

The Nearest Neighbor Algorithm approximates a TSP problem solution by selecting a city as the starting point, and then choosing the nearest city not already part of the graph. This chosen city is then used to find the next nearest city. This is repeated until the last city in the graph is chosen. The length of this graph is compared with the previous best and if better, gets stored as the current solution. The entire process above is then repeated for every city in the graph, using each of them as the start point for the next graph. This is an approximation graph, but has complexity of O(n^2)

```
temp
1       bestGraph = null
2       For every city startingCity in graph
3           thisGraph = startingCity
4           previousCity = startingCity
5   unusedCities = allCities - startingCity
6       While unusedCities.size > 0
7       nextCity = findNearestCity(previousCity)
8       unusedCities -= nextCity
9       previousCity = nextCity
10          thisGraph += nextCity
11      If thisGraph < bestGraph
12      bestGraph = thisGraph
13  Return bestGraph
```

Held-Karp

The Held-Karp algorithm is a method to solve the TSP which gives an exact solution, it utilizes the fact that every sub path of a path of minimum distance is itself of minimum distance. In order to calculate the best path using the Held-Karp algorithm we basically utilize the same method as Brute-force, and calculate all possible solutions. However, the Held-Karp algorithm is able to save time by utilizing the fact that we only need to calculate the remaining solutions of size n-1, after we've added a new vertex to the solution set. This means for each edge we add to the path, the size of the problem left to solve decreases by one. This gives the Held-Karp algorithm a slight time advantage over brute force which is O(n!), and puts it in the O(2^n n^2) range, due to its time

Hunter Land
Tanner Rousseau
Cody Ray

savings during the calculation of the remaining solutions. Because of this the algorithm uses subsets of solutions to store the best path, once all subsets of size two have been solved, the algorithm uses the calculated answers to construct a path, using each subset as a solution.

*S is an array to hold the best solution for each subset
*v, u are arbitrary vertices
*sol_arr holds id's of chosen cities

```
temp
1
2   TSPHeld-Karp()
3       Initialize 2d distance array
4       Start at arbitrary vertex
5       For every other vertex
6                   Dist_arr = distance(v1, vn)
7       For I = 2 to n
8                   For each subset S
9                                   For each v in S
10              For each u in S
11                                                  tmp = S[u][v] + distance_arr[u][v]
12                  If tmp < S[u][v]
13                      S[u][v] = z
14                      Sol_arr = u
15  Return path in sol_arr
```

Ant System

Ant Colony Optimization algorithms (ACOs) are algorithms inspired, as the name suggests, by the behavior of real ant colonies. These kinds of algorithms rely on a form of indirect communication between ants via pheromone trails. The algorithm of present interest is Ant System, which was the first ACO algorithm. This algorithm has been found to give poor solutions to larger problem instances, and a good amount of research has been geared toward improving Ant System, but the algorithm is an interesting and minimally complicated first step into ACOs.

The Ant System algorithm creates solutions for the Travelling Salesman Problem (TSP) by utilizing a set of *m "*ant" objects placed randomly at the various vertices (cities) in G.V. These ants create solutions by travelling to different cities, with the choice of where to go next being decided by a probability calculation based on levels of "pheromones,"

Hunter Land
Tanner Rousseau
Cody Ray

essentially higher ant "traffic," on a given edge from the current vertex to vertices in the feasible neighborhood (set of unvisited vertices) of the ant in question. The probability with which ant *k* chooses to go from city *i* to city *j* at the *t*th iteration is given by:

$$p_{ij}^k(t) = \frac{[\tau_{ij}(t)]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in \mathcal{N}_i^k} [\tau_{il}(t)]^\alpha \cdot [\eta_{il}]^\beta} \qquad \text{if } j \in \mathcal{N}_i^k$$

ACO Algorithms for the Traveling Salesman Problem

where d[i][j] is the distance between city *i* and city *j*, and n[i][j] = 1 / d[i][j] is a heuristic value to be scaled by *beta*. The quantities T[i][j](t) and T[i][l](t) are raised to *alpha*, and the quantities n[i][j] and n[i][l] are raised to *beta*. *Alpha* and *beta* are given parameters that determine the relative influence of the "pheromone trail" and the "heuristic information" on the determination of which vertex *l* in N (the set of vertices that ant *k* has not visited) that the ant will travel to next. Obtaining a sufficiently optimal solution depends on the tuning of these parameters, *alpha* and *beta*. If *alpha* is minimized, the algorithm essentially becomes a greedy algorithm, with the ants more likely to travel to the cities closest to their current city. If *beta* is minimized, the only thing determining an ant *k*'s next decision is the magnitude of the pheromone level on a given edge in N, which causes emergent behavior which gives unacceptably suboptimal solutions.

Based on this formula, we can see that any ant *k* will probabilistically prefer to travel to a city *l* in N in which the edge from *k*'s current city *i* to *l* has a shorter distance and a higher concentration of pheromone relative to any other city *l* in N.

In order to create plausible solutions, the ants need to be able to store their current partial tour. This is used at each tour construction step (iteration) to figure out which cities will be in N (the set of cities that ant *k* has not yet visited in the current tour. That is, the cities in *k*'s partial tour will be excluded from N.

After the tours have been created, the "pheromone trails" need to be updated by lowering the strength, or concentration, of ant pheromone trails on each edge by some constant factor and then having the ants deposit new pheromones on the edges in the partial tour they are currently creating. The update is done so that the edges that are either in the shorter tours and/or have been visited by more ants will have a higher probability of being chosen in the next iteration. The pheromone level of a given edge on the *t*th iteration is given by:

$$\tau_{ij}(t+1) = (1-\rho) \cdot \tau_{ij}(t) + \sum_{k=1}^{m} \Delta\tau_{ij}^k(t)$$

Hunter Land
Tanner Rousseau
Cody Ray

ACO Algorithms for the Traveling Salesman Problem

where *p* (the amount of pheromone evaporation) is a number between 0 and 1. This parameter is used so there will not be a build-up of pheromone on every edge, allowing the algorithm to exclude suboptimal routes.

The amount of pheromone that an ant deposits on the edges it has visited is given by:

$$\Delta\tau_{ij}^{k}(t) = \begin{cases} 1/L^{k}(t) & \text{if arc } (i,j) \text{ is used by ant } k \\ 0 & \text{otherwise} \end{cases}$$

ACO Algorithms for the Traveling Salesman Problem

where L[k](t) is the length of the *k*th ant's tour at the *t*th iteration. This causes more pheromone to be deposited to edges travelled by many ants, raising the probability of well-travelled edges being chosen in future iterations. After a sufficient number of iterations, an approximate optimal tour is arrived at by choosing the most well-travelled edges.

CS325 Project Report - Group 56

Hunter Land
Tanner Rousseau
Cody Ray

```
func ACO
    set parameters
    initialize pheromone trails
    while all cities not visited
        construct solutions
        apply local search
        update pheromone trails
    end
end

func ACO(G(V, E), alpha, beta, p, t, numAnts, numCities)
    initialize T[numCities][numCities] and P[numCities][numCities] with 0s
    // T will keep track of the pheromone levels for each edge in G.E
    // P will keep track of the probabilities of ants choosing an edge in G.E
    create n[numCities][numCities]
    create numAnts ants.k
    randomly place all ants at nodes in G.V
    for iteration = 0 to t
        for city in numCities
            for each ant k in ants.k do
                for each targetCity in k.N
                    n[k.currentCity][targetCity] = 1 / distance(k.currentCity, targetCity)
                    currentCalculation =  ((T[k.currentCity][targetCity]^alpha) * (n[k.currentCity][targetCity]))
                    runningCalculation += currentCalculation
                    P[k.currentCity][targetCity] = currentCalculation / runningCalculation
                    k.N.p.push(P[k.currentCity][targetCity])
                end
                k.currentCity = maxIn(k.N.p)
                k.currentTour.push(k.currentCity)
                T[k.currentCity][targetCity] = (1 - p) * T[k.currentCity][targetComputer] + runningCalculation
            end
        end
    end
    for each ant k in ants.k do
        k.bestTour = min(k.tours)
    end
    bestTour = min(ants.k.bestTour)
end
```

Discussion

Our group came to the conclusion that we should implement the Nearest Neighbor algorithm based on a few decisions. The first being that compared to both the Held-Karp algorithm and the Ant Colony method, Nearest Neighbor provides both an accurate and timely solution. While the Held-Karp algorithm provides an exact optimal solution, Nearest Neighbor is able to provide a nearly optimal solution while running in O(n^2),

Hunter Land
Tanner Rousseau
Cody Ray

while the Held-Karp algorithm can only achieve O(n^2 2^n). This difference in runtime was significant enough for us to conclude that the Held-Karp algorithm would not provide a timely solution. The second conclusion which led us to implement Nearest Neighbor is that the Ant Colony method requires a far more complicated implementation to run efficiently. The overhead to be able to implement an effective Ant Colony optimization was far greater than that required by the Nearest Neighbor algorithm based on our research.

```
temp
1   tour bestTour;
2       for (city startingCity in graph) {
3           tour thisTour;
4           thisTour.addCity(startingCity);
5           while (unvisitedCities) {
6               nextCity = findNearestCity();
7               thisTour.addCity(nextCity);
8           }
9       if (thisTour.length < bestTour.length) {
10          bestTour = thisTour;
11      }
12  }
13  return bestGraph;
```

Best Tours

All results are obtained from a modified nearestNeighbour algorithm which generates a maximum of 150 tours and selects the best of them.

Input file	Length	Time
tsp_example_1.txt	134,394	0.025s
tsp_example_2.txt	2990	0.862s
tsp_example_3.txt	1,925,701	20m31.746s

All results in the table below are generated from the nearestNeighbour algorithm, with a tour limit of 150.

CS325 Project Report - Group 56

Hunter Land
Tanner Rousseau
Cody Ray

Input file	Length	Time
test-input-1.txt	6,091	0.011s
test-input-2.txt	8,090	0.044s
test-input-3.txt	14,939	0.586s
test-input-4.txt	19,851	1.503s
test-input-5.txt	27,713	5.990s
test-input-6.txt	40,128	21.388s
test-input-7.txt	61,849	2m13.180s

Citations:

R. Bellman, “Dynamic Programming Treatment of the Travelling Salesman Problem,” *Journal of the ACM (JACM), vol. 9, no. 1, pp. 61–63,* 1962.

M. Held and R. M. Karp, “A Dynamic Programming Approach to Sequencing Problems,” *Proceedings of the 1961 16th ACM national meeting on -*, 1961.

St, Thomas & Dorigo, Marco. “ACO Algorithms for the Traveling Salesman Problem”, 1999.

B. Li, L. Wang, and W. Song, “Ant Colony Optimization for the Traveling Salesman Problem Based on Ants with Memory,” *2008 Fourth International Conference on Natural Computation*, 2008.