Thomas Hart
Cara Hamilton
Benjamin Harden
C S 312 sec 2
13 April 2021

# Traveling Salesman Project

Abstract :

The purpose of this paper is to discuss a few different approaches to solving the Traveling Salesman Problem and even implement our own solution to this problem. The implementations covered are the greedy approach, the branch and bound approach, and our own best solution. For our “fancy” solution we used a version of the Ant Colony Algorithm which we will discuss in further detail. We will start by covering the theoretical time and space complexities of each implementation. Then, using the data produced by each method, we will compare both the efficiency of each solution and also the resulting path of each solution. Our results show that both our own method and the branch and bound method produce a more optimal path than the greedy approach.

The purpose of the Traveling Salesman Problem is, given a list of cities and the paths between those cities, to find the minimum cost path of visiting every city once. We start with a random solution which does not use any techniques to produce an optimal solution. We then wrote the greedy solution which operates by always taking the immediate best choice. Our next best solution uses the branch and bound method to produce a more optimal solution. Finally, we implemented our own solution with the goal to produce a more optimal solution than the greedy approach.

Greedy:

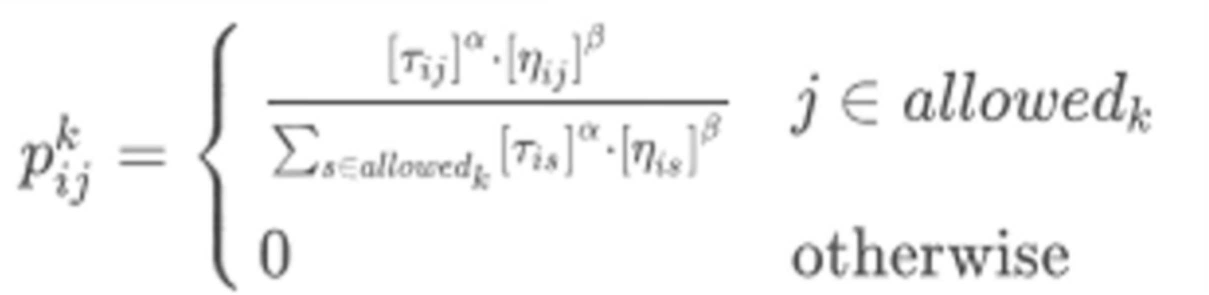
The greedy implementation of the TSP works by starting at a random city and always going to the next city by choosing the immediate shortest distance until it reaches all the cities. This method finds a fast solution but it is not always a very optimally low cost solution.
The time complexity is O(n^2) as each city needs to be compared to every other city.

The space complexity is O(n) for the lists containing the cities.

Fancy -- Ant Colony Algorithm:

The solution we came up with to solve the TSP is a version of an ant colony algorithm. It mimics how ants release pheromones in a certain manner, which allows other ants to follow the best path.

Each available edge from one vertex to another has a “probability” associated with it. The lowest probability edge available from the current vertex is chosen. What is unique about the ant colony algorithm is that the probability associated with edges is dynamic and constantly changing. We calculate it by using the distance, the pheromone level of that edge (which evaporates over time, but increases each time an ant takes that path), and some constant exponential factors we decide for ourselves. The specific formula is here:



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

“Tij” is the pheromone level from vertex i to vertex j. “Nij” is 1 divided by the distance. Alpha and Beta are constants, which we concluded were very efficient at α = 1 and β = 5. The denominator of the formula is the sum of possible probability values from vertex i to all other reachable vertices.

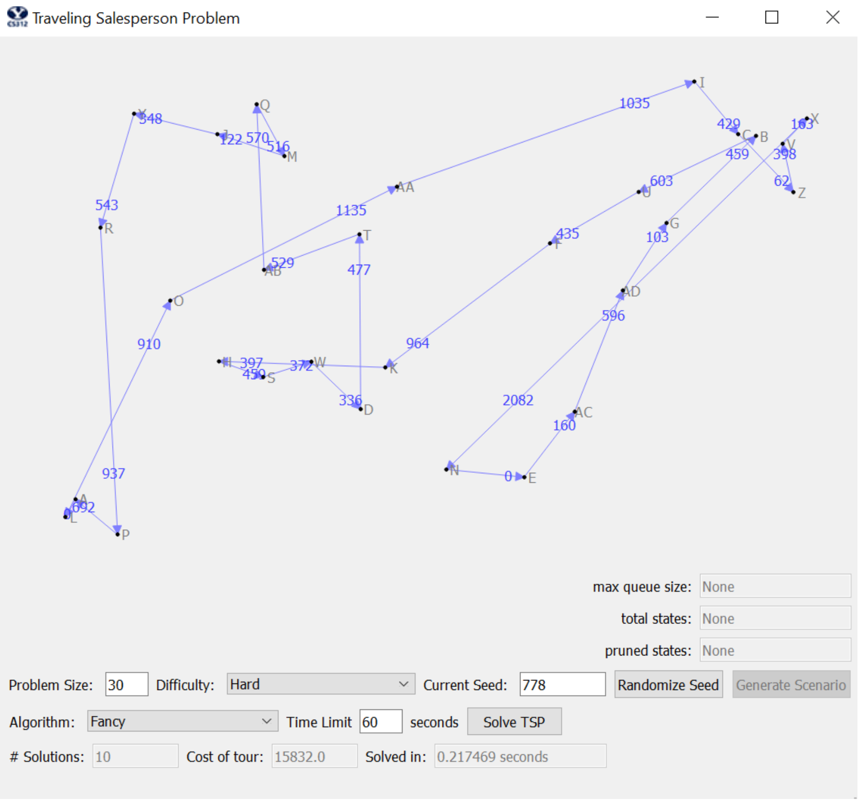
Each iteration in the “while” loop represents a new ant starting at a random vertex and finding a new path based upon the updated probabilities. The probabilities are updated as a result of the evaporating and/or increasing pheromone levels of each edge. (In the hard (deterministic) mode, these edges are not the same back and forth, so the pheromone level is not the same going from i to j and j to i.) The pheromone level for each edge decreases by half of its current value each iteration, and it is increased (1 divided by the current total path distance) if the ant took that edge during that iteration. After so many iterations, the pheromone trail should theoretically cause the ants to find the optimal solution. Our own contributions to the algorithm were the involvement of matrices that helped keep track of pheromone level, distances, etc. We also chose the constants that change the influence of distance and pheromone on the path the ants take.

Through empirical analysis, the algorithm does not always find the optimal solution, although it is much better than the greedy algorithm and much faster than branch and bound. There are mixed results depending on how many “ants” are sent to find a path and the constant values controlling probabilities and evaporation of pheromone. The time it takes and the solution found may be better or worse depending on the specific problem. Overall, this algorithm is a good balance between finding a good solution and finding a solution quickly. We can change how many ants we send to get a more optimal solution at the cost of speed and vice versa.

The time complexity is about O(n^3). For our implementation, the while-loop goes until time runs out or until it goes through 10 iterations. There are then 2 more nested for-loops (n iterations each), and O(n) work at the deepest level to find probabilities, update matrices, etc. This adds up to about O(n^3), plus some other lower time complexities for other functions and multiplied by the constant factor of 10. We originally tried having the while loop go through n iterations as more iterations would theoretically give a better solution. This was true, but the massive speed up seemed more worth it than a path with a better cost of about 1000 or so.

The space complexity is O(n^2). We use five n^2 matrices to keep track of the adjacency matrix, the temporary adjacency matrix with updated values, the pheromone levels for each edge, and two other matrices that help us keep track of and update the pheromone levels. We also use some lower space complexity lists for keeping track of the current path, best path, and results.

Examples:



Traveling Salesperson Problem

max queue size: None

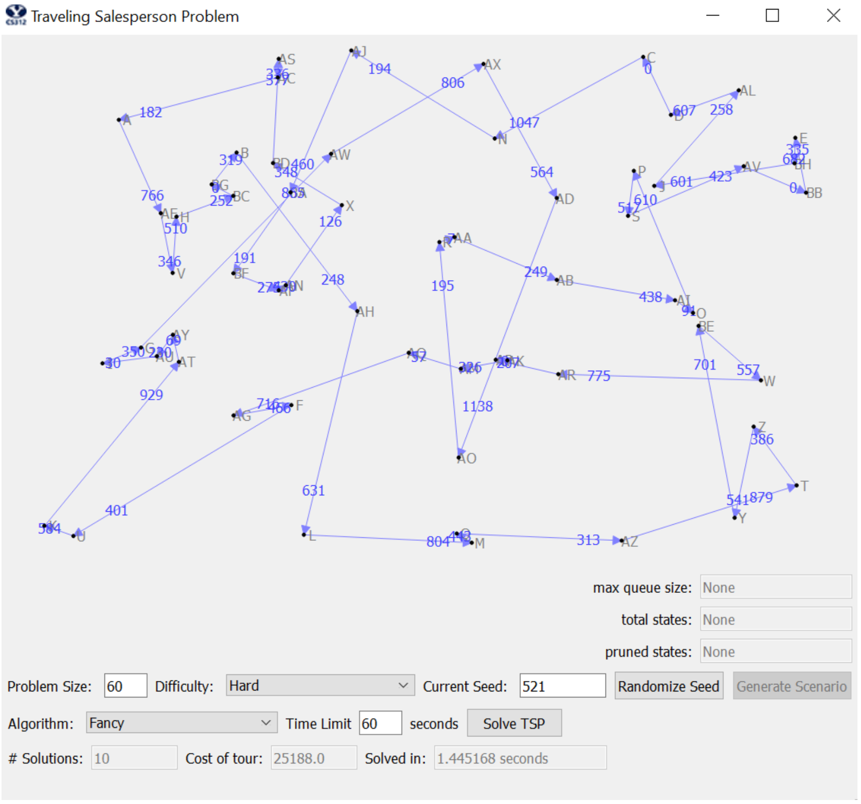
total states: None

pruned states: None

Problem Size: 30 Difficulty: Hard Current Seed: 778 Randomize Seed Generate Scenario

Algorithm: Fancy Time Limit 60 seconds Solve TSP

# Solutions: 10 Cost of tour: 15832.0 Solved in: 0.217469 seconds



Traveling Salesperson Problem

max queue size: None

total states: None

pruned states: None

Problem Size: 60 Difficulty: Hard Current Seed: 521 Randomize Seed Generate Scenario

Algorithm: Fancy Time Limit 60 seconds Solve TSP

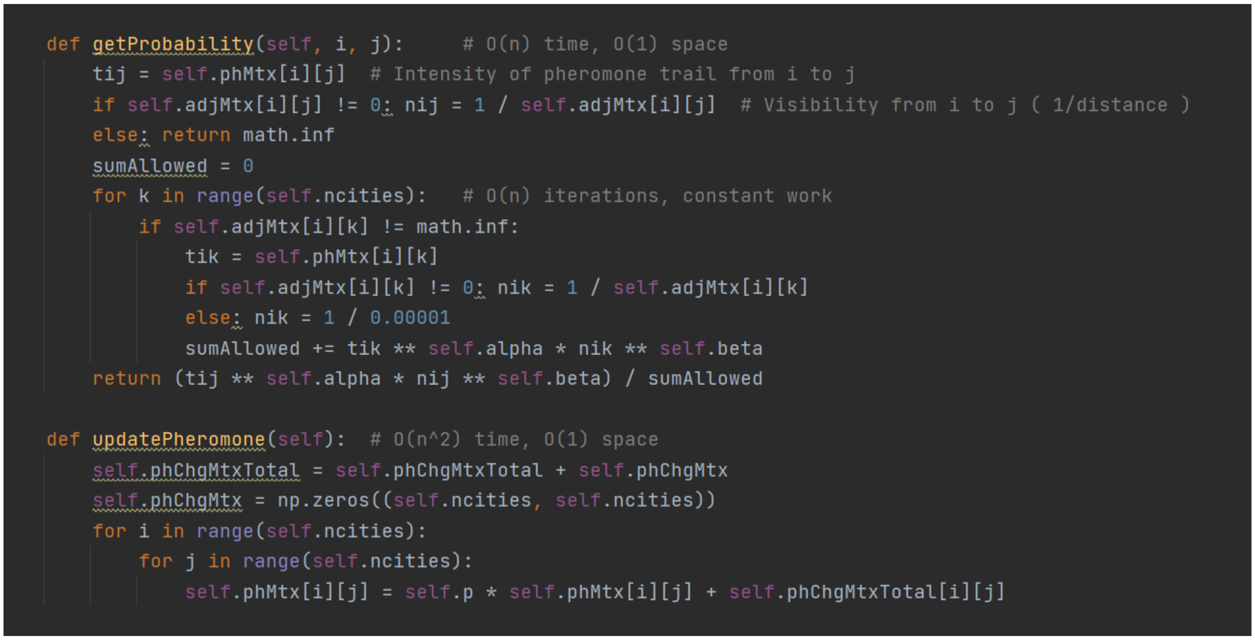
# Solutions: 10 Cost of tour: 25188.0 Solved in: 1.445168 seconds

Results

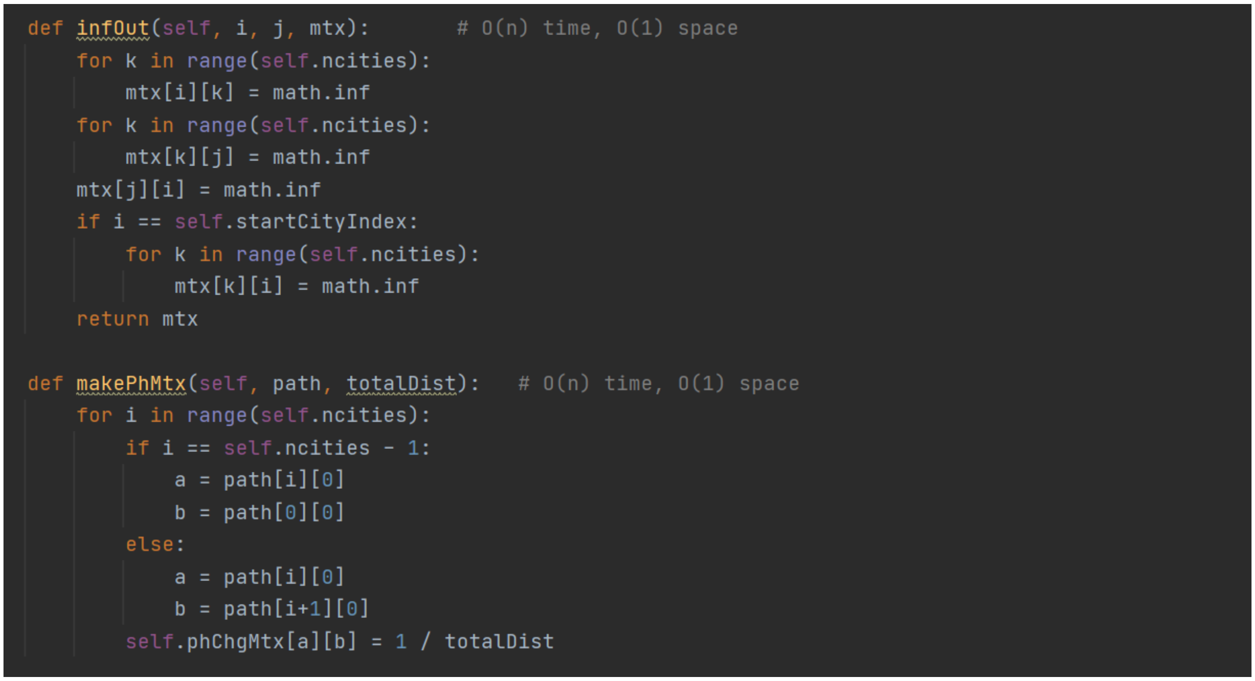
| | Random | | Greedy | | | Branch and Bound | | | Our Algorithm | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| # of cities | Time (sec) | Path length | Time (sec) | Path length | % of random | Time (sec) | Path length | % of greedy | Time (sec) | Path length | % of greedy |
| 15 | 0 | 21876 | .0045 | 10675 | 48.8% | 27.28 | 8868 | 83.1% | .034 | 9262 | 86.8% |
| 30 | 0 | 36919 | .0328 | 17124 | 46.4% | TB | TB | TB | .078 | 13861 | 80.9% |
| 60 | TB | TB | .2609 | 25753 | NA | TB | TB | TB | 0.75 | 21637 | 84% |
| 100 | TB | TB | 1.373 | 36379 | NA | TB | TB | TB | 8.76 | 33732 | 92.7% |
| 200 | TB | TB | 14.379 | 60359 | NA | TB | TB | TB | 41.24 | 55521 | 92% |
| 90 | TB | TB | .9854 | 33286 | NA | TB | TB | TB | 5.75 | 30557 | 91.8% |
| 95 | TB | TB | 1.1646 | 35989 | NA | TB | TB | TB | 5.31 | 33025 | 91.7% |
| 105 | TB | TB | 1.6187 | 38644 | NA | TB | TB | TB | 8.42 | 34071 | 88.1% |
| 110 | TB | TB | 1.8899 | 36264 | NA | TB | TB | TB | 9.7 | 35132 | 96.8% |
| 300 | TB | TB | 59.354 | 74040 | NA | TB | TB | TB | 74.21 | 71465 | 96.5% |

Our results show that the greedy solution gave a more optimal solution than the random solution as we expected. Our random solution could only solve a problem size for up to 30 cities. Our greedy approach resulted in an average cost of 47.6% of the random approach. The branch and bound approach and also our “fancy” algorithm gave more optimal solutions than the greedy approach. The branch and bound approach could not solve for problem sizes higher than 20, but compared to greedy on a size of 15, the branch and bound approach gave a cost of 83.1% of the greedy method. For our algorithm, we are able to solve for a pretty large number of cities relatively fast. Our algorithm did not always come up with the most optimal solution, however it did still result in a shorter path than the greedy solution. The “sweet spot” for our solution was around a problem size of 100 cities. At this size our algorithm would solve in around 5 seconds and it would result in a cost of around 91% of the greedy approach.

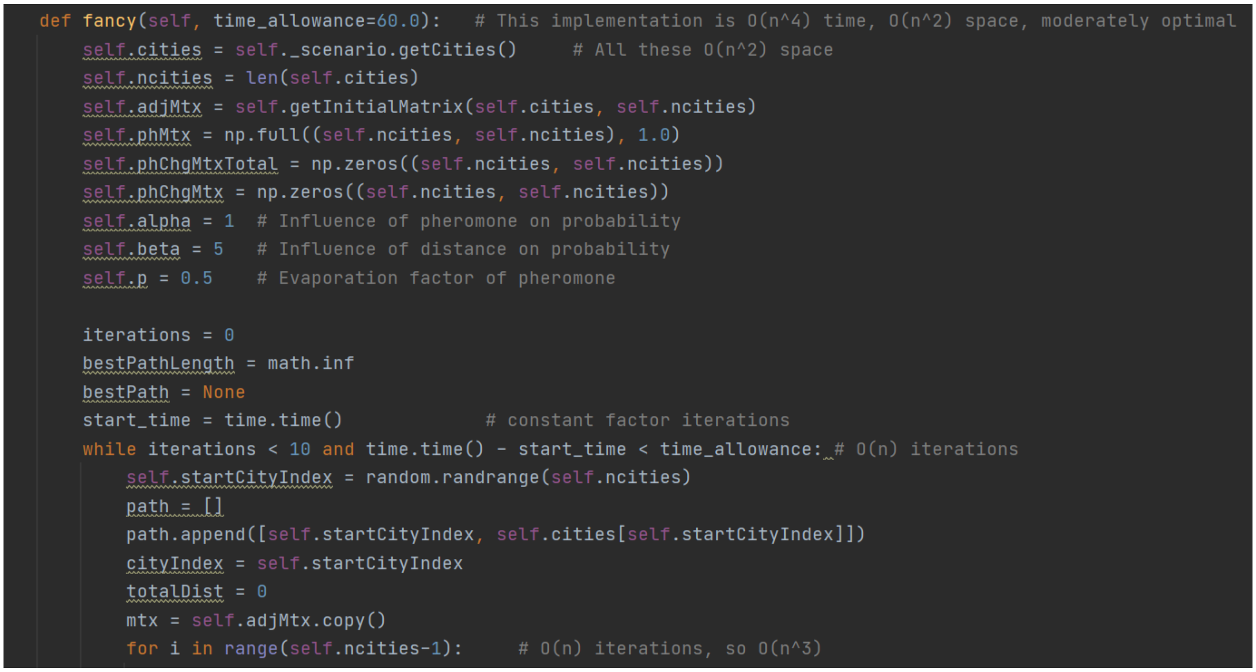
Code for Fancy algorithm:



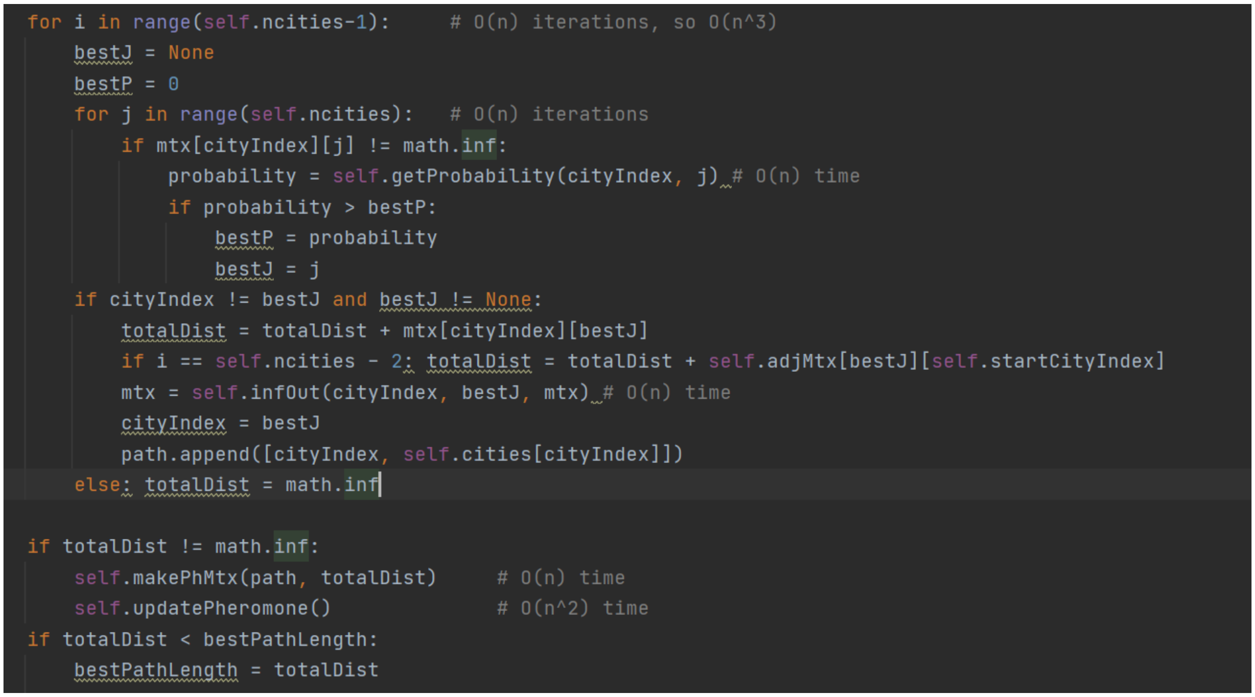
```python
def getProbability(self, i, j):     # O(n) time, O(1) space
    tij = self.phMtx[i][j]  # Intensity of pheromone trail from i to j
    if self.adjMtx[i][j] != 0: nij = 1 / self.adjMtx[i][j]  # Visibility from i to j ( 1/distance )
    else: return math.inf
    sumAllowed = 0
    for k in range(self.ncities):   # O(n) iterations, constant work
        if self.adjMtx[i][k] != math.inf:
            tik = self.phMtx[i][k]
            if self.adjMtx[i][k] != 0: nik = 1 / self.adjMtx[i][k]
            else: nik = 1 / 0.00001
            sumAllowed += tik ** self.alpha * nik ** self.beta
    return (tij ** self.alpha * nij ** self.beta) / sumAllowed

def updatePheromone(self):  # O(n^2) time, O(1) space
    self.phChgMtxTotal = self.phChgMtxTotal + self.phChgMtx
    self.phChgMtx = np.zeros((self.ncities, self.ncities))
    for i in range(self.ncities):
        for j in range(self.ncities):
            self.phMtx[i][j] = self.p * self.phMtx[i][j] + self.phChgMtxTotal[i][j]
```



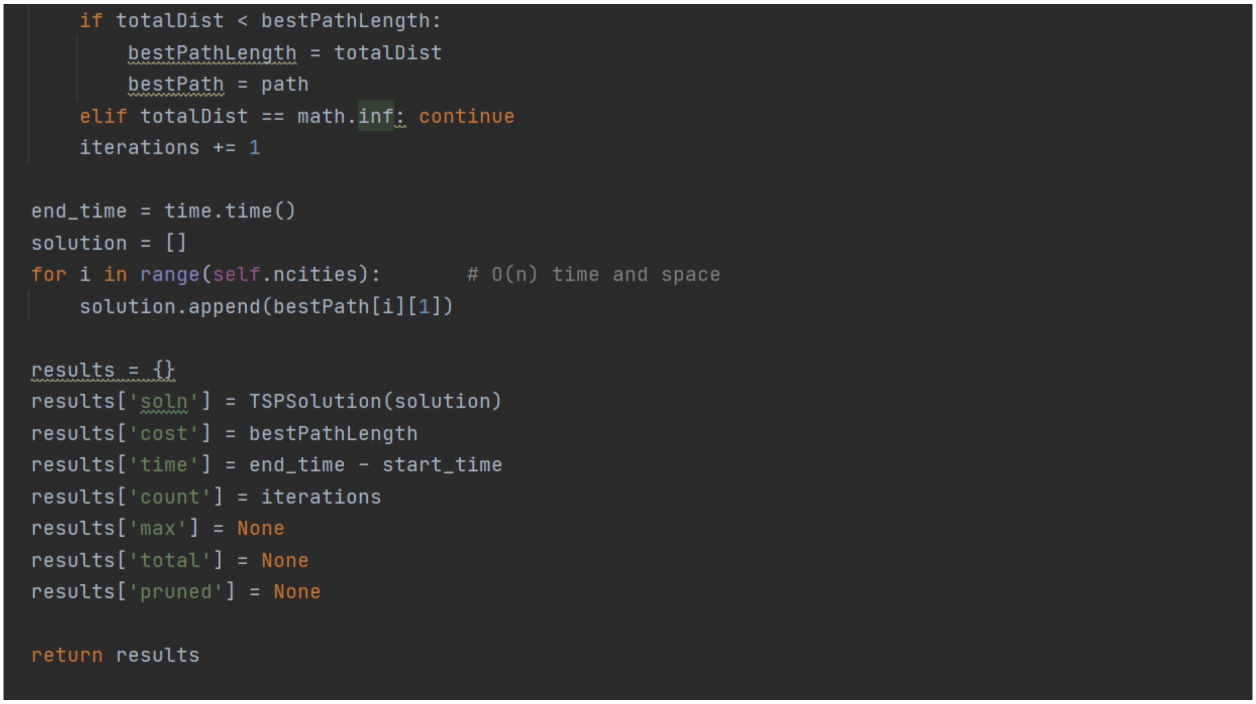
```python
def infOut(self, i, j, mtx):       # O(n) time, O(1) space
    for k in range(self.ncities):
        mtx[i][k] = math.inf
    for k in range(self.ncities):
        mtx[k][j] = math.inf
    mtx[j][i] = math.inf
    if i == self.startCityIndex:
        for k in range(self.ncities):
            mtx[k][i] = math.inf
    return mtx

def makePhMtx(self, path, totalDist):   # O(n) time, O(1) space
    for i in range(self.ncities):
        if i == self.ncities - 1:
            a = path[i][0]
            b = path[0][0]
        else:
            a = path[i][0]
            b = path[i+1][0]
        self.phChgMtx[a][b] = 1 / totalDist
```



```
def fancy(self, time_allowance=60.0):   # This implementation is O(n^4) time, O(n^2) space, moderately optimal
    self.cities = self._scenario.getCities()     # All these O(n^2) space
    self.ncities = len(self.cities)
    self.adjMtx = self.getInitialMatrix(self.cities, self.ncities)
    self.phMtx = np.full((self.ncities, self.ncities), 1.0)
    self.phChgMtxTotal = np.zeros((self.ncities, self.ncities))
    self.phChgMtx = np.zeros((self.ncities, self.ncities))
    self.alpha = 1  # Influence of pheromone on probability
    self.beta = 5   # Influence of distance on probability
    self.p = 0.5    # Evaporation factor of pheromone

    iterations = 0
    bestPathLength = math.inf
    bestPath = None
    start_time = time.time()             # constant factor iterations
    while iterations < 10 and time.time() - start_time < time_allowance: # O(n) iterations
        self.startCityIndex = random.randrange(self.ncities)
        path = []
        path.append([self.startCityIndex, self.cities[self.startCityIndex]])
        cityIndex = self.startCityIndex
        totalDist = 0
        mtx = self.adjMtx.copy()
        for i in range(self.ncities-1):     # O(n) iterations, so O(n^3)
```

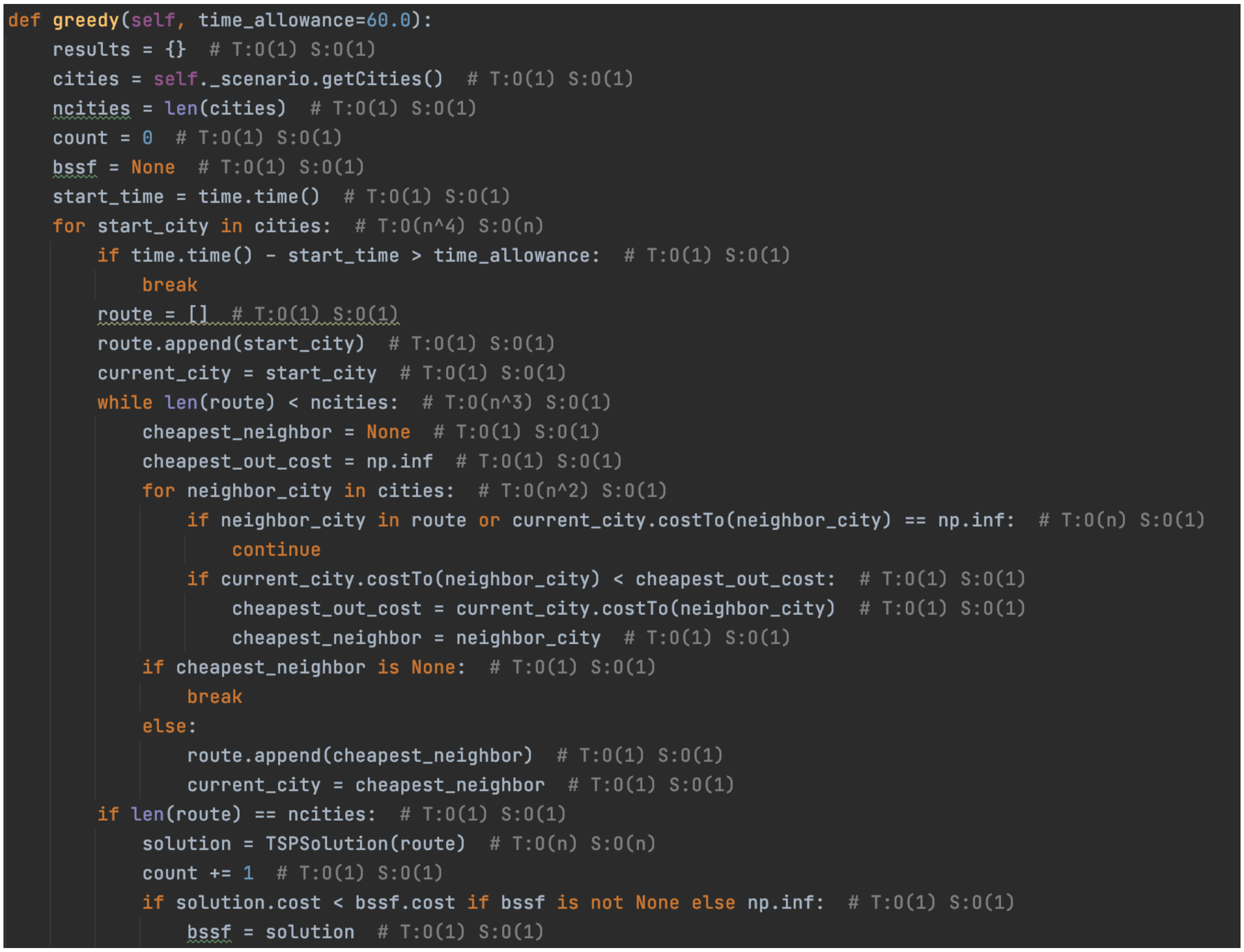


```
for i in range(self.ncities-1):     # O(n) iterations, so O(n^3)
    bestJ = None
    bestP = 0
    for j in range(self.ncities):   # O(n) iterations
        if mtx[cityIndex][j] != math.inf:
            probability = self.getProbability(cityIndex, j) # O(n) time
            if probability > bestP:
                bestP = probability
                bestJ = j
    if cityIndex != bestJ and bestJ != None:
        totalDist = totalDist + mtx[cityIndex][bestJ]
        if i == self.ncities - 2: totalDist = totalDist + self.adjMtx[bestJ][self.startCityIndex]
        mtx = self.infOut(cityIndex, bestJ, mtx) # O(n) time
        cityIndex = bestJ
        path.append([cityIndex, self.cities[cityIndex]])
    else: totalDist = math.inf

if totalDist != math.inf:
    self.makePhMtx(path, totalDist)     # O(n) time
    self.updatePheromone()              # O(n^2) time
if totalDist < bestPathLength:
    bestPathLength = totalDist
```

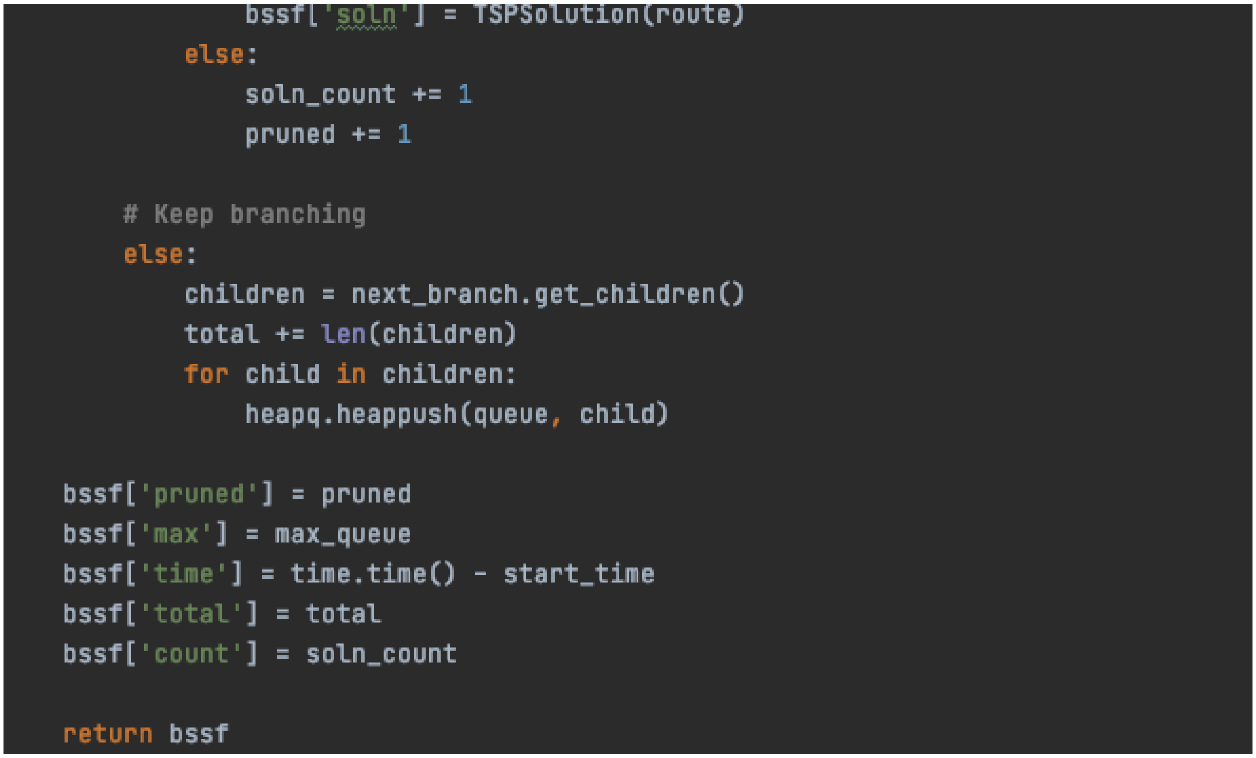


```
    if totalDist < bestPathLength:
        bestPathLength = totalDist
        bestPath = path
    elif totalDist == math.inf: continue
    iterations += 1

end_time = time.time()
solution = []
for i in range(self.ncities):       # O(n) time and space
    solution.append(bestPath[i][1])

results = {}
results['soln'] = TSPSolution(solution)
results['cost'] = bestPathLength
results['time'] = end_time - start_time
results['count'] = iterations
results['max'] = None
results['total'] = None
results['pruned'] = None

return results
```

Code for Greedy algorithm:



```
def greedy(self, time_allowance=60.0):
    results = {}  # T:O(1) S:O(1)
    cities = self._scenario.getCities()  # T:O(1) S:O(1)
    ncities = len(cities)  # T:O(1) S:O(1)
    count = 0  # T:O(1) S:O(1)
    bssf = None  # T:O(1) S:O(1)
    start_time = time.time()  # T:O(1) S:O(1)
    for start_city in cities:  # T:O(n^4) S:O(n)
        if time.time() - start_time > time_allowance:  # T:O(1) S:O(1)
            break
        route = []  # T:O(1) S:O(1)
        route.append(start_city)  # T:O(1) S:O(1)
        current_city = start_city  # T:O(1) S:O(1)
        while len(route) < ncities:  # T:O(n^3) S:O(1)
            cheapest_neighbor = None  # T:O(1) S:O(1)
            cheapest_out_cost = np.inf  # T:O(1) S:O(1)
            for neighbor_city in cities:  # T:O(n^2) S:O(1)
                if neighbor_city in route or current_city.costTo(neighbor_city) == np.inf:  # T:O(n) S:O(1)
                    continue
                if current_city.costTo(neighbor_city) < cheapest_out_cost:  # T:O(1) S:O(1)
                    cheapest_out_cost = current_city.costTo(neighbor_city)  # T:O(1) S:O(1)
                    cheapest_neighbor = neighbor_city  # T:O(1) S:O(1)
            if cheapest_neighbor is None:  # T:O(1) S:O(1)
                break
            else:
                route.append(cheapest_neighbor)  # T:O(1) S:O(1)
                current_city = cheapest_neighbor  # T:O(1) S:O(1)
        if len(route) == ncities:  # T:O(1) S:O(1)
            solution = TSPSolution(route)  # T:O(n) S:O(n)
            count += 1  # T:O(1) S:O(1)
            if solution.cost < bssf.cost if bssf is not None else np.inf:  # T:O(1) S:O(1)
                bssf = solution  # T:O(1) S:O(1)
```



```
            bssf['soln'] = TSPSolution(route)
        else:
            soln_count += 1
            pruned += 1

    # Keep branching
    else:
        children = next_branch.get_children()
        total += len(children)
        for child in children:
            heapq.heappush(queue, child)

bssf['pruned'] = pruned
bssf['max'] = max_queue
bssf['time'] = time.time() - start_time
bssf['total'] = total
bssf['count'] = soln_count

return bssf
```