Project Report - CS7.507. Multi-Agent Systems

I. Base Paper

Description

The paper we have chosen is **"A scalable multi-robot task allocation algorithm"** by Chayan Sarkar, Himadri Sekhar Paul and Arindam Pal of TCS Research and Innovation, Kolkata. The paper can be found here.

In modern warehouses, robots can be deployed to fetch a set of objects having certain weights from various locations all across the warehouse and bring them back to a docking station. Each robot can carry multiple objects on a single run, and is constrained by a capacity, above which it cannot carry any weight, i.e. the combined weight of the objects that the robot carries cannot exceed this capacity. The goal is to allocate items to certain robots and decide optimal routes for the robot to collect these items such that the cost (time to complete, or distance) to travel these routes is minimized.

This is an instance of the **Capacity-Constrained Vehicle Routing Problem (CVRP)**, which is an NP-hard problem. There exist a number of exact and heuristic solutions to this problem, but when the number of items increases (to the order of thousands), then most of these solutions become highly inefficient in terms of cost, runtime and number of routes. Simply put, existing solutions do not scale very well with the number of items.

The paper proposes a heuristic, which they call **nearest neighbour-Based Clustering and Routing (nCAR)**, which performs much better compared to the state-of-the-art heuristics, and gives a solution comparable to the optimal one even when the number of items reaches a high order. The state-of-the-art solution, Google's OR-Tools has 1.5 times the execution time and number of routes of nCAR, and nCAR has a runtime speedup of 6 when the item size is 2000.

Problem Formulation

We have a:

- Set of **robots** that can carry maximum weight of C (capacity). The robots are assumed to be homogeneous, and so all of them have the same capacity.
- Set of **objects/nodes** with certain weights

- **Warehouse** where these objects are located
- **Dock** to which these objects should be brought

The goal is to assign objects/tasks to robots such that the overall cost to bring all objects to the dock is minimized.

Mathematical Formulation

In order to translate this problem to computational terms and determine an efficient algorithm to solve this problem, we represent the warehouse and dock as a complete, unweighted graph, with the nodes being the items, the edges between them and their costs representing the distances between the items, and the 0th node representing the dock.

In other words,

- We have a graph $G = \{V, E\}$ with $|V| = n + 1, V = \{v_0, v_1, ..., v_n\}$.
- Each edge has a cost $c > 0$.
- Each vertex v_i has an associated demand $d_i, d_i > 0$.
- C is the maximum capacity of each robot and $C > d_i$ for all i.

The goal is to find t cycles C_i, such that:

- $\sum_{i=1}^{t} c(C_i)$ is minimized (minimizing total cost of routes).
- $C_i \cap C_j = \{v_0\}$ for $1 \leq i < j \leq t$ (the cycles/routes found must be vertex-disjoint, i.e. no robot can pick up items assigned to another robot).
- $\sum_{v_j \in C_i} d_j \leq C$ for $1 \leq i \leq t$ (no robot can carry more than its capacity).

$c(C)$ is the cost of a cycle.

II. Key Contributions of Paper

- **Fast and Scalable Heuristic:** The paper proposes nCAR, an O(n^3)-time heuristic algorithm, which gives a solution, that given the number of items and a representation of the distances between the items, tells us the minimal cost, number of robots required and the routes for each robot
- **Improved Cost for More Nodes:** Cost of the solution given by nCAR is:
  - at most 1.6 times the optimal solution for benchmark datasets with smaller number of nodes
  - significantly better than the SOTA algorithm for test datasets with larger number of nodes
- **Improved Number of Routes & Runtime:** Number of routes computed by nCAR to complete all tasks (fetch all items) of a CVRP instance is near-optimal, and outperforms the SOTA solution with respect to the required number of routes and time required to find a solution, irrespective of the number of nodes in the CVRP instance.

III. Algorithm Implementation

Base Method

The algorithm has been implemented using Python. The code has been split across 2 files: `ncar.py`, which contains the core of the implementation of the nCAR algorithm, and `tsp.py`, which contains methods specific to the cluster routing part of the algorithm.

The input of the nCAR algorithm is:

- A set of nodes/items, their demands (weights) and the distances between the items

- Capacity that each robot must have

We represent this set as a graph, in turn represented by an adjacency matrix containing the edge weights between i and j as $graph[i][j]$. The function `nCAR` contains the implementation of the algorithm, and returns:

- a set of routes for each robot
- number of robots required
- total cost to travel all routes - the minimal cost

```
def nCAR(capacity: int, graph: List[List[int]], weight: List[int]):
    for node in range(len(graph)):
        if weight[node] > capacity:
            print("Error: Weight of node ", node, " exceeds capacity")
            sys.exit(0)

    routes = []
    robot_count = 0
    total_cost = 0

    unassigned_nodes = list(range(1,  len(graph)))

    while len(unassigned_nodes) > 0:
        assigned, remaining = feasible_route(
            unassigned_nodes, capacity, graph, weight)
        route, cost = travelling_salesman(assigned, graph)

        routes.append(route)
        total_cost += cost
        robot_count += 1

        unassigned_nodes = remaining
        print("Unassigned nodes: ", unassigned_nodes)

    return routes, robot_count, total_cost
```

The nCAR algorithm is divided into 2 parts:

1. **Cluster Assignment:** Split nodes into clusters containing disjoint sets of nodes, following these criteria:
    a. Total demand of nodes within cluster is within capacity of robot.
    b. Clusters are formed such that their individual cost leads to lower total path cost.
2. **Cluster Routing:** Construct an optimal route for each cluster formed.

Cluster Assignment

This part of the algorithm performs the task allocation - deciding which robots are to pick up which items. It has been implemented in the method `feasible_route`.

```
def feasible_route(
    unassigned_nodes: List[int], capacity: int, graph: List[List[int]], weight: List[int]
):
    p_min = single_node_cycle(unassigned_nodes, graph) + 1
    assigned = []
    remaining = []

    for current_node in unassigned_nodes:
        cluster = [0, current_node]
        non_cluster = unassigned_nodes.copy()
        non_cluster.remove(current_node)
        cluster_weight = weight[current_node]
        active_node = current_node

        while cluster_weight < capacity:
            k = restricted_nearest_neighbour(
                active_node, non_cluster, cluster_weight, capacity, graph, weight
            )

            if k == 0:
                break

            cluster_weight += weight[k]
            cluster.append(k)
            non_cluster.remove(k)
            active_node = k

        p = euler_cycle(cluster, graph) + single_node_cycle(non_cluster, graph)
        if p < p_min:
            p_min = p
            assigned = cluster
            remaining = non_cluster

    return assigned, remaining
```

Essentially, what `feasible_route` does is form a set of potential or “candidate” clusters, and choose the best cluster out of them. It does this repeatedly for a group of nodes, dividing them into clusters until every node has been assigned to a cluster. Each cluster formed is then assigned to a robot. Therefore, 2 tasks are accomplished here: task allocation and discovery of number of robots required.

The cluster assignment algorithm is as follows:

1. Initially, we start with all nodes as unassigned (not part of any cluster).
2. If there are N unassigned nodes, we create N potential clusters for each node. The i-th node is by default part of the i-th cluster.
   a. For each cluster, we take the last node added to the cluster to be the “current node". We then find the nearest neighbour of the current node in terms of Euclidean distance (basically the edge weight in this case) with the condition that the weight of this cluster should not make the total weight of the cluster exceed the robot’s capacity. This is done by the method `restricted_nearest_neighbour`.
   b. If no such neighbour is found, we terminate this operation, and the cluster is formed.
   c. If a neighbour is found, we add it to the cluster, and make it the “current node”.
3. Now that we have all N potential clusters formed, we must determine the best among them. For this, we employ a greedy approach, assigning a cost to each cycle. We then take the cycle with the minimum cost. This cost is equal to the summation of:
   a. Cost of **Euler cycle** of the nodes in the cycle. An Euler cycle is a trail which starts and ends at the same vertex and visits each edge exactly once. This cost is obtained by the method `euler_cycle`.
   b. Summation of costs of all **single node cycles** of unassigned nodes outside the cluster (in the set of unassigned nodes that we are considering). A single node cycle is a a cycle from node 0 (the dock) to a particular node and back. This cost is obtained by the method `single_node_cycle`.

This method of allocation, with the added cluster cost heuristic, ensures that both current and potential future cluster optimality are considered.

Cluster Routing

Now that we have a set of optimal clusters allocated to each robot, our task is to construct a route for the robot - essentially determine the order in which the robot should collect these items to minimize the cost of travel (the distance travelled). Since the robot only has to visit the item once, this becomes an instance of the **Travelling Salesman problem**, which is NP-hard.

There are several heuristic algorithms to approach this problem. The best known approximation is the **Christofides approximation**, which has an approximation ratio of 1.5, and is what the paper uses.

The Christofides algorithm is composed of 5 steps:

1. **Create a minimum spanning tree (T) of the given graph (G)** (the subgraph formed by the obtained cluster). This can be done using Prim's algorithm or Kruskal's algorithm, and for our implementation we have chosen to use Kruskal's algorithm, done in `minimum_spanning_tree`).
2. **Make a set (S) of vertices having an odd degree in (T).** The degree of a node is the number of edges connected to it. The number of these nodes would be even. This has been implemented in `find_odd_vertexes`.
3. **Find a minimum-cost perfect matching (M) from (S).** Essentially, we try to connect pairs of vertices in (S) (similar to bipartite matching) and ensure that the connections (edges between connected nodes) are chosen such that the total edge weight is minimal.
4. **Add the set of edges from (M) to (T) to form a new graph (G')** where all vertices would have an even degree. Steps 3 and 4 have been implemented in `minimum_weight_matching`.
5. **Create an Eulerian cycle of (G')** (done in `find_eulerian_tour`) and remove all repeated vertices from it to find the optimal ordering (done in `remove_edge_from_matchedMST`).

The implementation of this algorithm is done by the method `travelling_salesman`, which references the methods in `tsp.py`, containing the full implementation of the Christofides algorithm.

```
def tsp(G):
    # Build a minimum spanning tree
    MSTree = minimum_spanning_tree(G)

    # Find odd vertexes
    odd_vertexes = find_odd_vertexes(MSTree)

    # Add minimum weight matching edges to MST
    minimum_weight_matching(MSTree, G, odd_vertexes)

    # Find an Eulerian tour
    eulerian_tour = find_eulerian_tour(MSTree, G)

    current = eulerian_tour[0]
    path = [current]
    visited = [False] * len(eulerian_tour)
    visited[eulerian_tour[0]] = True
    length = 0

    # Remove repeated nodes
    for v in eulerian_tour:
        if not visited[v]:
            path.append(v)
            visited[v] = True
            length += G[current][v]
            current = v

    length += G[current][eulerian_tour[0]]
    path.append(eulerian_tour[0])
    index = path.index(0)
    shifted_path = []

    if index != 0:
        path.pop()
        for i in range(0, len(path)):
            shifted_path.append(path[(i + index) % len(path)])
        shifted_path.append(0)
    else:
        shifted_path = path

    return shifted_path, length
```

Time Complexity Analysis

As stated earlier, one of the advantages nCAR has over the SOTA solution is its better runtime capabilities. It runs in O(n^3) time.

`feasible_route` iterates through all unassigned nodes in order to create a cluster for each of them. In the worst case. assuming a cluster of one node is formed in each pass, then the cost of this operation is O(n).

`euler_cycle` iterates through the nodes in the cluster, while `single_node_cycle` iterates through the unassigned nodes outside the cluster - essentially all the unassigned nodes together. Therefore the complexity of this computation is O(n). Since this is done within `feasible_route` for each potential cluster formed (the total number is equal to number of unassigned nodes), the cost of looping through the clusters now becomes O(n^2).

Assuming now that we form a cluster of one node each pass in the worst case, the for loop in the `nCAR` method is executed at most $|V|$ times, where V is the set of nodes - meaning that its complexity is O(n) as well. Since `feasible_route` is run each time by this loop, the overall complexity of `nCAR` is O(n^3).

IV. Results Obtained

The evaluation of nCAR's performance is done based on 3 metrics:

1. Cost of the solution
2. Number of routes in the solution
3. Execution time of the algorithm

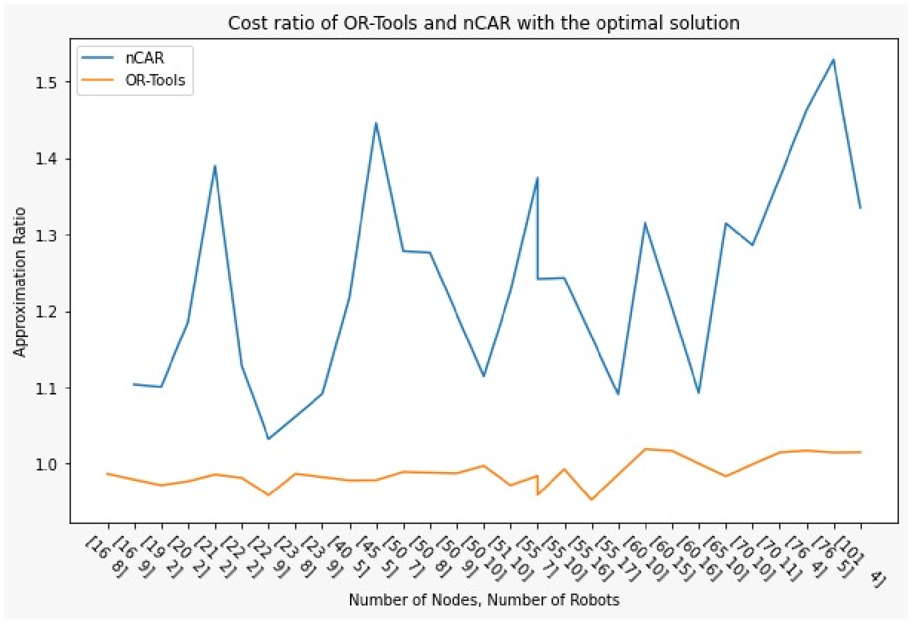
Our implementation was run on a standard system with an **Intel i5-9300H processor** and **16 GB of RAM**.

There already exist benchmark datasets containing CVRP instances with a number of nodes lesser than 1000, which we have listed in our results table below. The paper tested both the nCAR algorithm and the SOTA solution, Google's OR-Tools on these datasets and compared the results. We chose to adopt the same approach, testing our implementation and OR-Tools on these benchmark datasets. The results are described below.

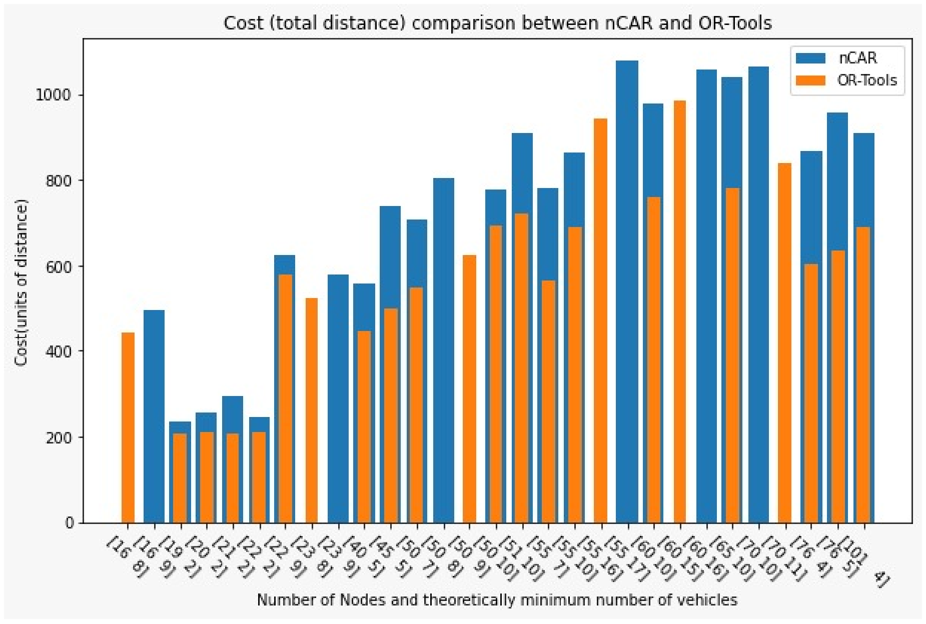
Smaller Datasets

Dataset	Maximum Routes	Capacity	OR-Tools Cost	OR-Tools Routes	nCAR Cost	nCAR Routes
P-n22-k8	8	3000	590	8	622.48	9
P-n23-k8	8	40	522	8	577.2	9
P-n45-k5	5	150	499	5	724.002	5
P-n50-k8	8	120	623	9	816.79	8
P-n55-k8	8	160	565	7	708.79	7
P-n60-k15	15	80	987	15	1055.59	16
P-n65-k10	10	130	779	10	1041.18	10

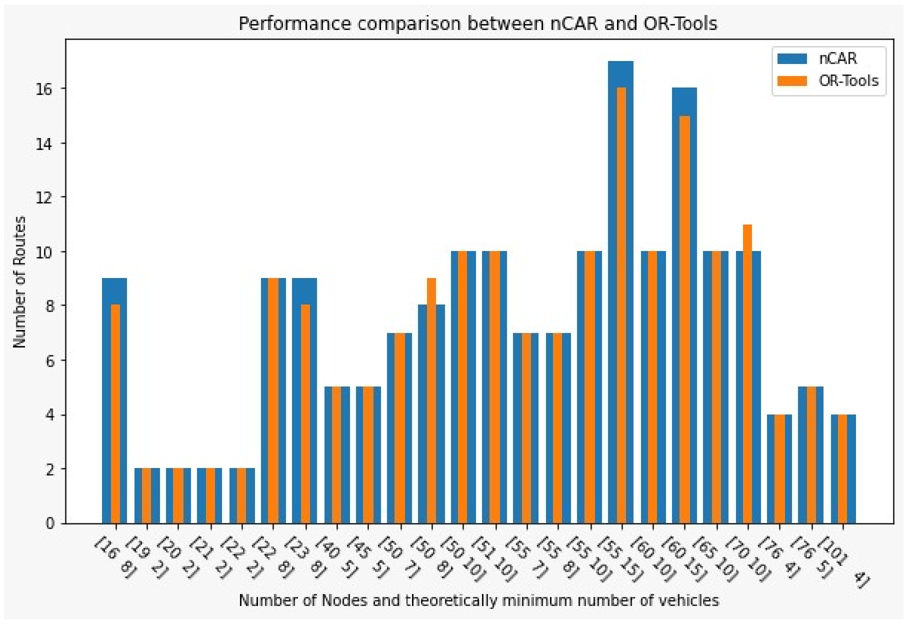
The P-datasets mentioned here are smaller datasets with node size being very small (the maximum node size in the above datasets is 101). Here, the results show that OR-Tools performs better than nCAR for smaller datasets (with node size $\leq$ 101). This should not be surprising, since OR-Tools is currently the SOTA solution and is expected to perform almost optimally. However, nCAR does not do poorly either. The number of routes is almost the same as that provided by OR-Tools, and the cost given by nCAR does not exceed 1.45 times that of OR-Tools (as shown by the paper)



The above graph shows similar results mentioned above (number of robots is essentially the number of routes found). Approximation ratio is the ratio of the value (cost or number of routes) of the heuristic to that of the optimal solution. While there does seem to be a bit of fluctuation in the approximation ratio for nCAR, the solution given by nCAR does not exceed 1.6 times the optimal solution. This shows that nCAR provides a good level of optimality comparable to both OR-Tools and the most optimal solution.

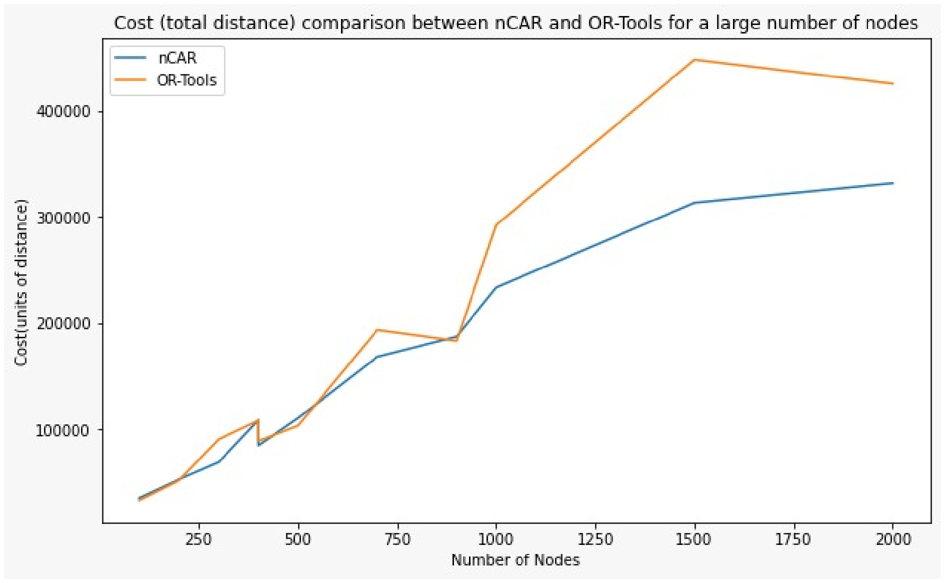


The above graph shows how the performance of the nCAR compares to the OR-tools in situations with a lower number of nodes. The maximum number of nodes in these graphs are 101. Here, we observe that OR-tools seem to outperform nCAR.



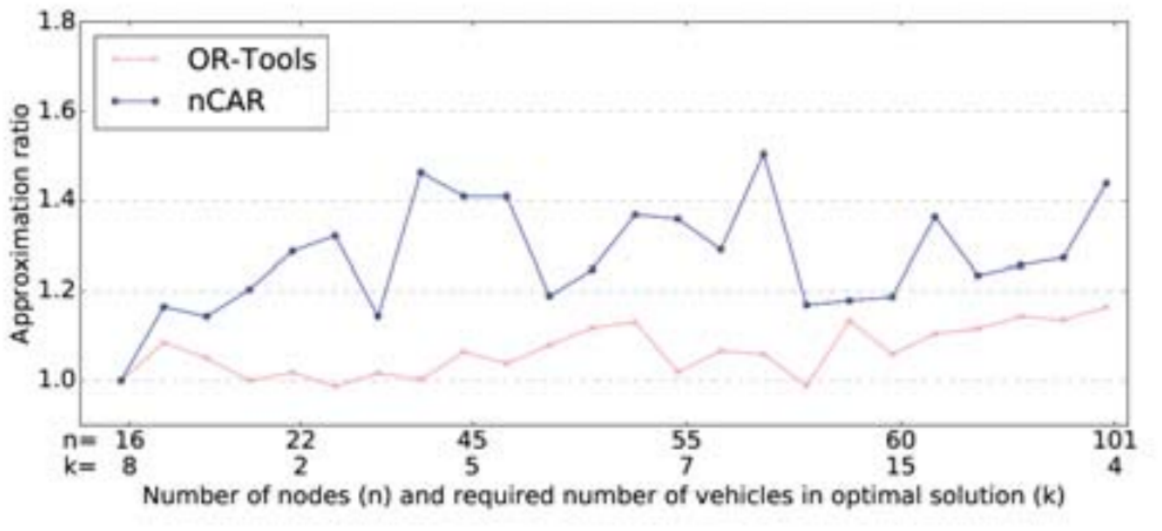
Again, here we observe that for smaller datasets(with lesser number of nodes) the number of routes taken in the solution is around the same for both nCAR and OR-tools.

Larger Datasets

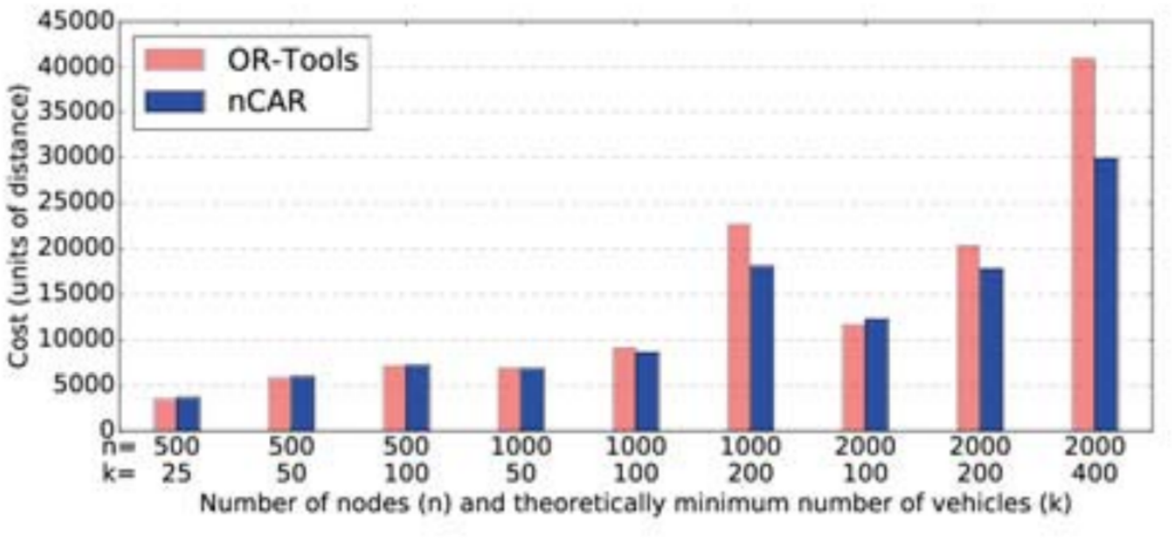


Even though OR-Tools performs slightly better than nCAR for small node sizes (less than 100), and slightly worse than nCAR for medium node sizes (between 250 and 1000), nCAR performs significantly better for node sizes above 1000, as evidenced by the plot above. This demonstrates the scalability of nCAR as node size reaches a higher order.

V. Our Results vs. The Paper's Results



The results we have obtained are similar to those obtained by the paper. Looking at the approximation ratio graph of the paper, we observe that similar to ours, the ratio does not seem to cross 1.6 as the theory dictates.



In situations with lower number of nodes, OR-tools does better than nCAR albeit with a small margin, but as the number of nodes increase, as is observed in our results, nCAR soon takes the lead as the better hueristic.

VI. Learnings From Project

This section reflects what we learned from understanding and implementing the paper "A scalable multi-robot task allocation algorithm".

The key challenges of multi-robot task allocation (MRTA): MRTA is a complex problem that involves assigning tasks to robots in a way that maximizes efficiency and minimizes cost. In our case, there was an added constraint satisfaction problem as well.

We studied how constraint satisfaction problems can model tasks and resources in a warehouse setting, and what are the challenges of solving them in a distributed and cooperative way, such as resource allocation, task coordination, uncertainty and dynamics.

We learned how to implement the nCAR approach in our project, using a graphical model to represent the problem, a graph partitioning algorithm to decompose the problem, a combinatorial auction algorithm to allocate subproblems to agents, and a search algorithm to execute subproblems by agents. This also involved studying how to generate Euler cycles for a set of nodes, and how to use the Christofides approximation to come up with an optimal route for a connected graph.

This project has helped our understanding in the direction of applying multi-agent systems to real-world problems and applications, such as service robots, transportation systems, exploration of hazardous environments, homeland security and rescue in disaster scenarios, among others. These are real world scenarios where intelligent multi-agent systems may be needed for their distributed and cooperative reasoning. By implementing and evaluating the nCAR approach, we have learned about the challenges and opportunities of multi-agent systems, such as resource allocation, task coordination, uncertainty and dynamics, and how to overcome them using novel algorithms and techniques. Further, we have also learned about the potential implications and extensions of your work for other domains that involve constraint satisfaction, such as scheduling, planning, and optimization.