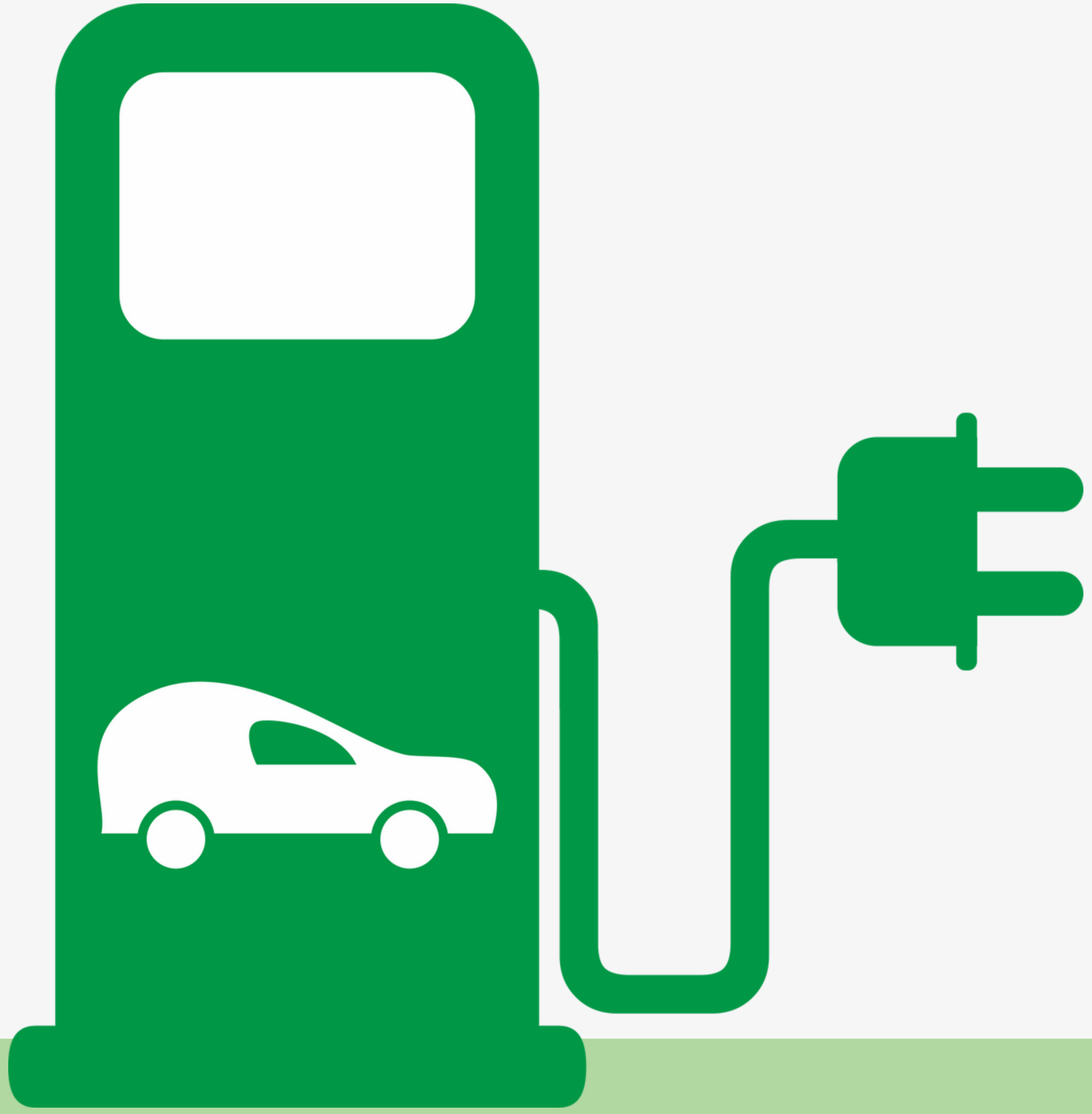
Qiskit Global Hackathon
2021



QAOA for smart charging of electric vehicles

Approaching industrial NP-hard problems

Our Motivation:

to work for a better future

Climate change is one of the greatest challenges of our time. In order to still reach the 1.5° target, all scientists must join forces and ask themselves how they can make a contribution with their research.

That is what we have done. In order to slow down climate change, electric cars will gain in importance in the future and with that their charging.

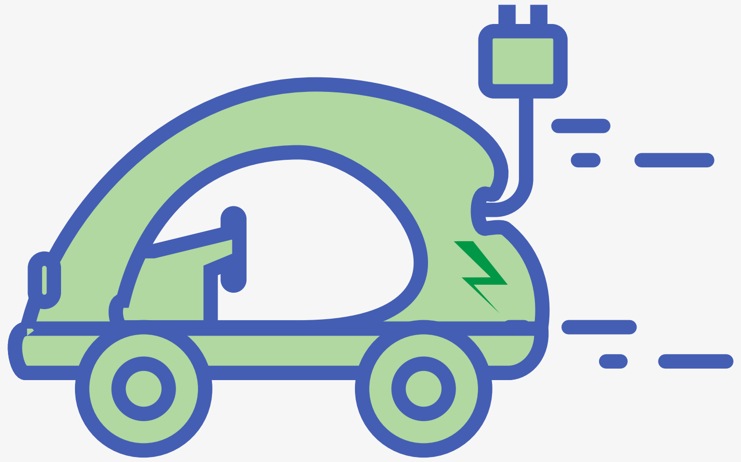
What is Smart Charging?

bidirectional charging

personalized charging

improve the flexibility of the electric system

Vehicle to grid



reducing high-peaks

batteries as energy storage and power supply

better time management

providing electicity if demand is high

2 Problems connected with Smart Charging

Minimization of Total Weighted Load Completion Time

Optimal Scheduling of Load Time Intervals within Groups

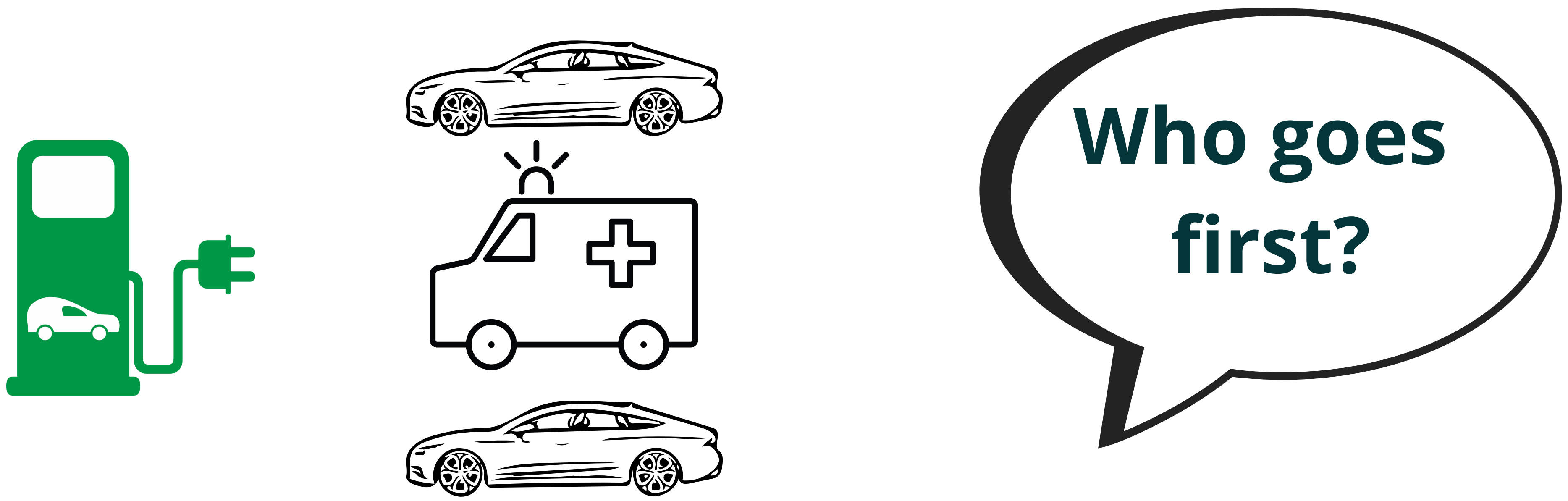
Our assumptions

In order to design a simple environment for solving the problems we make some restrictions to reality.

- load station is made up of several charging points
- each charging point can charge a single car at a given time step
- charging points supply same power
- charging time is independent of charging point
- no consideration of further job characteristics or global constraints
- load tasks can not be interrupted

Problem 1:
Minimization of Total Weighted Load Completion Time

J = {1, . . . , n}	: charging jobs
n	: electr. vehicles
T = {t1, . . . , tn}	: charging duration
I = {1, . . . , k}	: set of k charging points
wj > 0	: weight, measuring the importance
Cj	: completion time

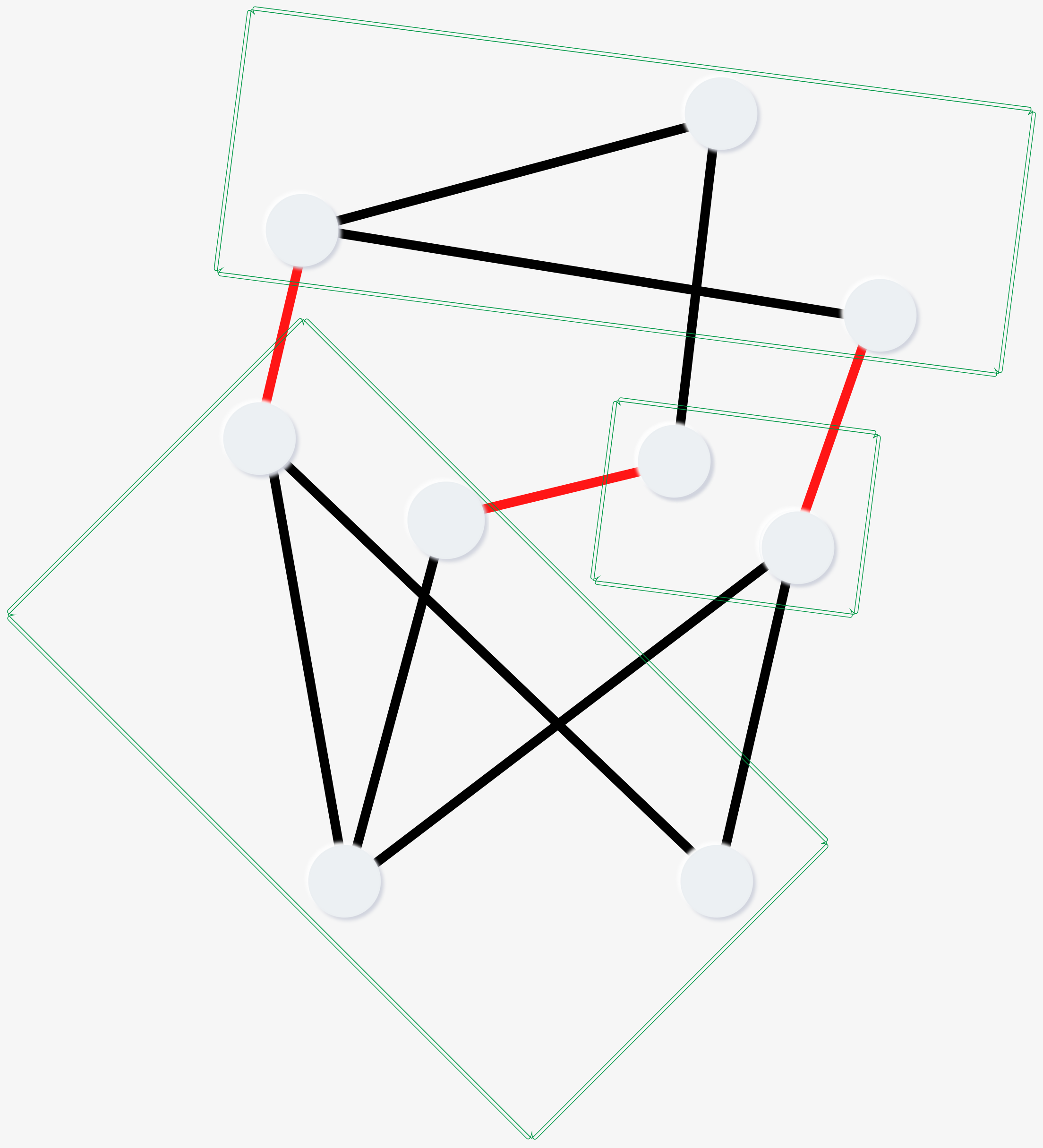


Task: minimizing the weighted total time of completion of the charges:

$$\sum_{j \in J} w_j C_j$$

Problem 1 is a Max-k-Cut problem

- Each *node* is a job with weight w and takes time t to complete, each edge between nodes i and j is min{w_i*t_j, w_j*t_i}
- w_i*t_j is the cost, which incures by putting job j before i (it's proportional to the importance of job i and to the time lost for i)
- If an edge in our plot is short, it has a huge cost; none of the node jobs should wait for the other -> vehicles should be sent to different charging ports
- Applying Max-k-Cut gives *k connected subgraphs* by cutting *short edges*
- The nodes of each subgraph represents vehicles that are in the same queue. The loading order is determined by non-increasing order of w / t.



Problem 1: Implementation

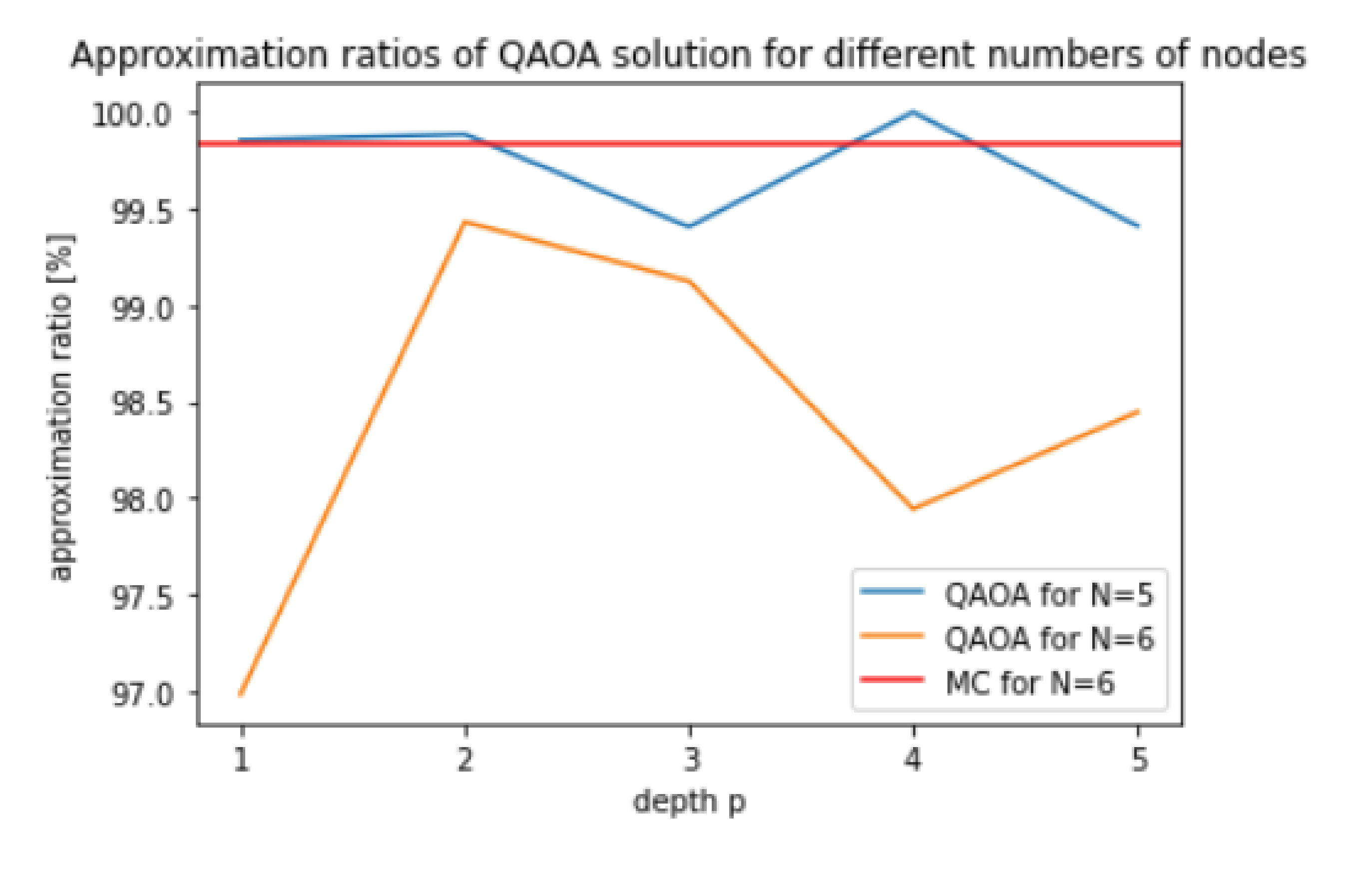
Data Creation: We used networkx to design a graph. We created random charging times and weights for n cars. After that we assigned cost values to the connections between two cars.

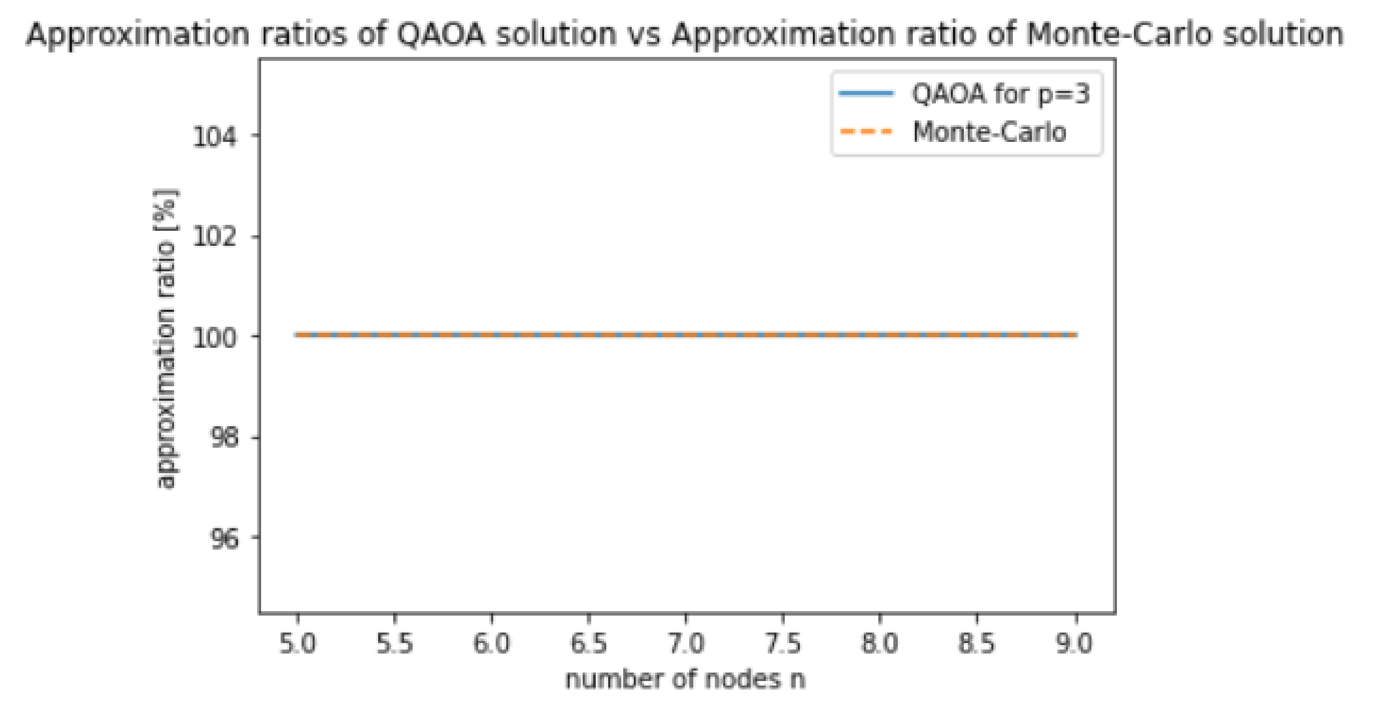
Max-k-Cut Implementation: We translate the cost appearing in the Max-k-Cut problem into a Hamiltonian. For this we can find the ground state energy using a combined loop of quantum computing combined and classical optimization. Check out our Jupyter Notebook for more detail!

Benchmarking: We implemented a classical Monte-Carlo-Solver to compare the QAOA results to a classical solution.

Problem 1: Results

- Our QAOA solver is working and gives solid approximation ratios over 95%.
- We only tested on small data sets since the computation time is quite high.
- From the upper graph one can guess, that a higher depth will improve the results.



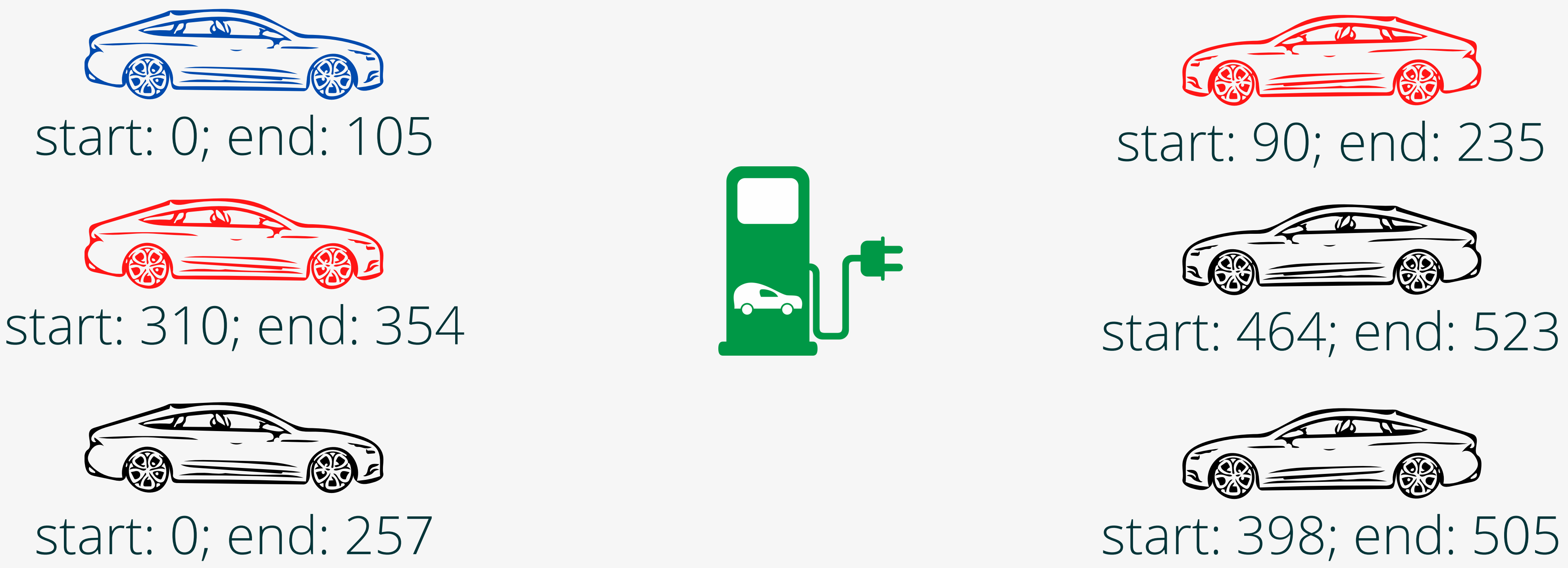


Problem 2: Optimal Scheduling of Load Time Intervals within Groups

I = {(s_1, e_1) (s_xn, e_xn)}: set of intervals (load job start- and end- date)

x : number of groups

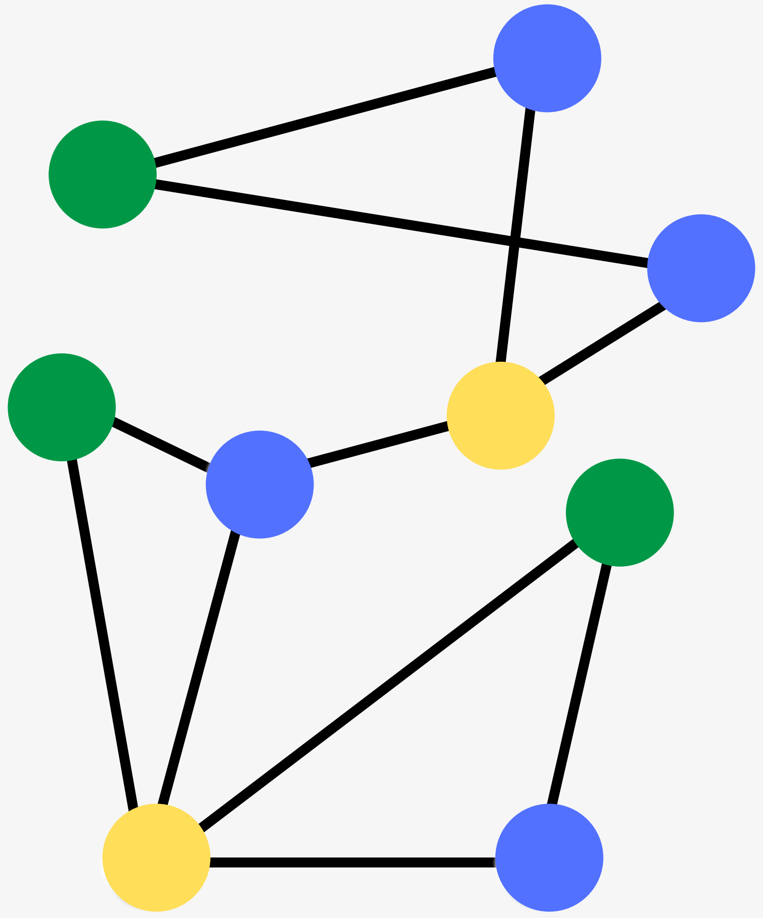
n : number of tasks



Tasks: Which loading tasks should be handled, so that no group (colour) is overpresented (max. one load in each group) and that the number of non-overlapping tasks is maximal?

Problem 2: The optimal solution is a Maximum Independent Set

- A Maximum Independent Set (MIS) of a Graph G is an indepent subset of nodes, such that no two nodes, appearing in the subset, are directly connected by an edge.
- Each node represents a loading task, so each node has three attributes: group, start time and end time.
- There is a connection between two nodes if 1) the two nodes are in the same group or 2) the charging intervals are overlapping or 3) both is true.
- The goal is now to find a maximal subset of not directly connected nodes.



Problem 2: Implementation

Data Creation: We used networkx to design a graph. For k jobs we created random charging intervals and assign them to n groups. After that we connected cars within the same group and/or within the same time interval.

MIS Implementation: We translate the search for the MIS into calculating the minimum of a cost function. Again, we encode this cost function into a Hamiltonian and search for its lowest energy state by using a combined loop of quantum computing combined and classical optimization. Check out our Jupyter Notebook for more detail!

Problem 2: Results

The MIS algorithm is not working perfectly at the moment since it sometimes ignors the constraints. We have to go over it again! 🤨

```python
def compute_cost_MIS(counts, w, U, n_counts=512):
    """
    :param counts:  dict{measurement result in string:count}
    :param l: The number of qubits representing a node
    :param w: The adjacency matrix for edges
    :param U: The size of punishment added to the cost due to including nodes of the same group in the MIS
    :return:  The averaged cost
    """
    total_cost = 0
    for measurement, count in counts.items():
        preprocessed_chosen_set = (len(measurement)-1)-np.argwhere(np.array(list(measurement))=='1')
        if len(preprocessed_chosen_set) == 0:
            continue
        chosen_set = np.concatenate(preprocessed_chosen_set)
        total_cost += -1 * len(chosen_set) * count
        if len(chosen_set) < 2:
            continue
        else:
            for edge in combinations(chosen_set, 2):
                if w[edge[0]][edge[1]] == 1:
                    total_cost += U * count
    average_cost = total_cost / n_counts
    return average_cost
```

```python
def full_optimization_loop_MIS(n, w, U, p, bounds=[(-np.pi, np.pi), (0, 4*np.pi)], nshots=512,
                               simulator='qasm_simulator', local_optimization_method='BFGS', optimal_cost=None):

    ##############################################################################################################
    # Initialize
    backend = Aer.get_backend(simulator)
    backend.shots = nshots
    param_history = []
    cost_history = []
    circ_history = []
    ##############################################################################################################
    # Run the educated global guess (EGG) optimization for the first time
    circ = make_full_circuit_MIS(n, w, 1)
    circ_history.append(circ)
    func_to_optimize = func_to_optimize_wrapper_MIS(circ, w, U, nshots=nshots, simulator=simulator)
    result = differential_evolution(func_to_optimize, bounds)
    param, cost = result.x, result.fun
    print('1st params', param)
    print('1st cost', cost)
    param_history.append(param)
    cost_history.append(cost)
    ##############################################################################################################
    # If depth = 1, no need to continue
    if p < 2:
        return param_history, cost_history, circ_history
    ##############################################################################################################
    # Else continue
    for i in range(2, p+1):
        if i == 2:
            abbrev = 'nd'
        else:
            abbrev = 'th'
    ##############################################################################################################
    # Run the educated global guess (EGG) optimization for ith iteration
        circ = make_full_circuit_MIS(n, w, i)
        param_names = circ.parameters
        param_bind_dict = {}
        for j in range(i-1):
            param_prev = param_history[-1]
            param_bind_dict[param_names[j]] = param_prev[j]
            param_bind_dict[param_names[j + i]] = param_prev[j + i - 1]
        circ_w_param = circ.bind_parameters(param_bind_dict)
        circ_history.append(circ_w_param)
        func_to_optimize = func_to_optimize_wrapper_MIS(circ_w_param, w, U, nshots=nshots, simulator=simulator)
        result = differential_evolution(func_to_optimize, bounds)
        param, cost = result.x, result.fun
        complete_param = np.concatenate((param_prev[:i-1], np.array([param[0]]),
                                         param_prev[i-1:], np.array([param[1]])))
        print(str(i) + abbrev + ' iteration (EGG), params', complete_param)
        print(str(i) + abbrev + ' iteration (EGG),  cost', cost)
        param_history.append(complete_param)
        cost_history.append(cost)
    ##############################################################################################################
    # Run the local optimization of choice for ith iteration if needed
        if local_optimization_method is not None:
            func_to_optimize = func_to_optimize_wrapper_MIS(circ_w_param, w, U, nshots=nshots, simulator=simulator)
            result = minimize(func_to_optimize, complete_param, method=local_optimization_method)
            param, cost = result.x, result.fun
            print(str(i) + abbrev + ' iteration (' + local_optimization_method + '), params', param)
            print(str(i) + abbrev + ' iteration (' + local_optimization_method + '), cost', cost)
            param_history.append(param)
            cost_history.append(cost)
            circ_history.append(circ)
    ##############################################################################################################
    # If optimal cost found by brute force is provided, compute the approximation ratio evolution
    if optimal_cost is not None:
        print('Approximation Ratio Evolution ', cost_history / optimal_cost)
    ##############################################################################################################
    # lists of parameters, costs and quantum circuits are returned
    return param_history, cost_history, circ_history
```

Summary: What we have done

Implementation 1

Implementing an algorithm minimizing the total weighted load completion time & a benchmarking.

Visual Simulation

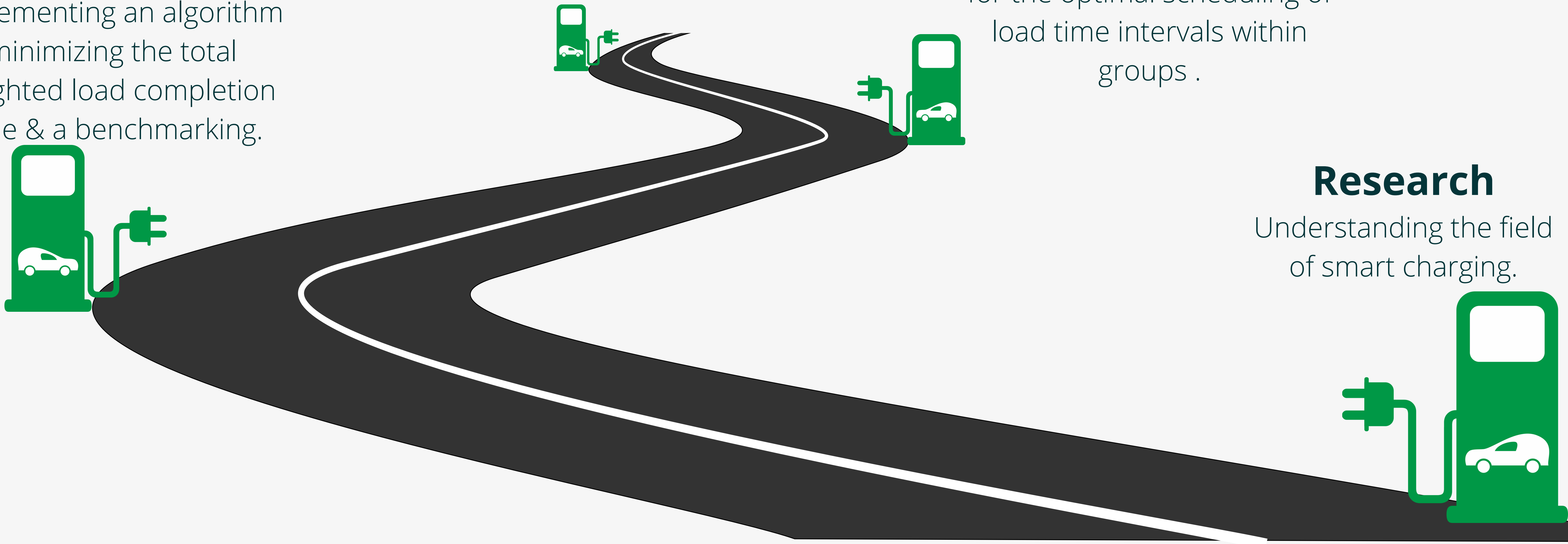
Creating a Django web app with a visual simulation of what our code does.

Implementation 2

Implementing an algorithm for the optimal scheduling of load time intervals within groups .

Research

Understanding the field of smart charging.



You want to see what our code does?

Have a look into our Visualization!

[Visualization]

You want to know more?

Have a look into our Jupyter Notebooks!

[More detail]

Next steps

Improving

We need to further improve the Max-k-Cut algorithm in order to make it faster.

We also need to look into our MIS algorithm again since it is not working proberly right know.

Extending

A next step would be to change the assumptions in order to make it more realistic. External factors, like the status of the electric grid, and interruptions should be caputured.

Developing

The long-term goal would be to design an app, where electric car users can enter the time, when they need their car again and for approximatly how many km. The app then calulates the best order to charge the cars by taking also external factors into account.

Our Team

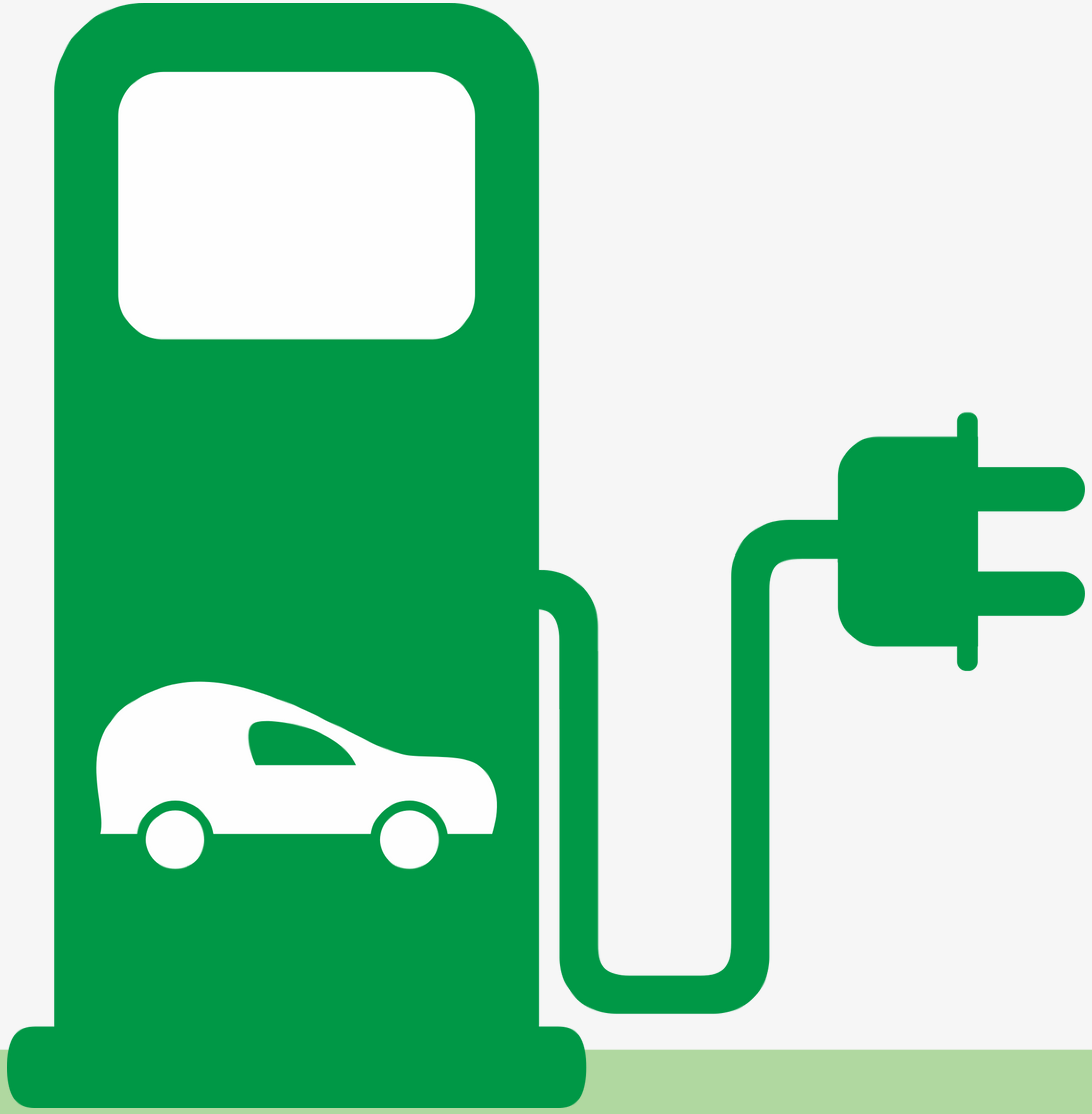
Kevin Shen
Quantum Science & Technology M.Sc.
Technical University of Munich

Catharina Broocks
Quantum Science & Technology M.Sc.
Technical University of Munich

Franziska Wilfinger
Quantum Science & Technology M.Sc.
Technical University of Munich

Jakob Pforr
Quantum Science & Technology M.Sc.
Technical University of Munich

JezerJojo
BSMS
IISER Pune



Thank you!

Resources

- https://arxiv.org/pdf/2012.14859.pdf
- https://arxiv.org/pdf/1411.4028.pdf
- https://arxiv.org/pdf/1709.03489.pdf
- https://www.osti.gov/servlets/purl/1756438
- https://qiskit.org/documentation/tutorials/optimization/6_examples_max_cut_and_tsp.html
- https://qiskit.org/textbook/ch-applications/qaoa.html