CAD2CAV: Computer Aided Design for Cooperative Autonomous Vehicles

Zhihao Ruan

***Abstract*—Autonomous robots have been put into use to explore unknown environments. With the development of a 1/10 scale race car, we have considered an application of using multiple F1/10 race cars to explore an unknown environment collaboratively while exploiting the prior knowledge of this environment, *i.e.,* the floor plan of the target building. We have proposed a complete pipeline from the building floor plan to 3D rendering of the building model and a planning stack that is capable of distribute the exploration task evenly to multiple vehicles.**

I. Introduction

In recent years, great improvements have taken place in terms of the technologies of autonomous vehicles and they have been put into use in various aspects. In particular, O'Kelly et al. [1], [2] have proposed an open-source simulation framework targeting autonomous racing cars at the 1/10 scale. In such context, this project considers a potential use for the F1/10 racing cars — to collaboratively explore an unknown environment efficiently given some prior knowledge, *i.e.,* exploring a building given its floor plans. More specifically, our challenge is to develop algorithms for the following steps: 1) set up a basic render of the 3D model of the building given its floor plan, 2) identify and extract some abstract waypoints from the floor plan, 3) split the waypoints evenly to all F1/10 racing cars so that each of them is able to explore a certain area of the building parallelly with minimum overlap and minimum total time, 4) navigate the car to explore the waypoints in real world, 5) use LiDAR to perceive the detailed information of the building and add these additional information to the rendering software.

We have come up with solutions to multiple steps in the task. For the rendering software of the building, we first use Autodesk Revit to redraw the floor plan using built-in architectural components, *i.e.,* walls, doors, windows, etc. Then we construct the 3D model of the building in Unreal Engine 4 (UE4) through a plugin called Unreal Datasmith, which can import the Autodesk Revit model into UE4 software; for the splitting of the waypoints for multiple F1/10 racing cars, we construct a graph connecting all the waypoints, apply multilevel k-way graph partitioning algorithms [3] to generate even-weight subgraphs, and then form a closed TSP (Travelling Salesman Problem) loop from each subgraph for each racing car; for the navigation of the racing cars, we use FMT* [4] to generate obstacle-avoiding paths for the racing cars to explore the waypoints, and use pure-pursuit [5] to control the racing cars; for the perception of the detailed information, we wish to make use of Google Cartographer [6] to perform SLAM in the environment and pass pre-processed point cloud information to the rendering software to update the model of the building.

II. Methods

A. The Rendering Software

The first step of this project is to set up a software that renders the basic model of the building from its floor plans. We approached this by making use of the existing model rendering plugin Unreal Datasmith[1]. Namely, given the building's floor plan in AutoCAD format, we first import it to Autodesk Revit and redraw the floor plan using built-in architectural components, so that the floor plan has its own 3D view in Autodesk Revit. Figure (1) demonstrates the result of redrawing the floor plan in Revit. Next, we export the model through Unreal Datasmith Revit Plugin, and import it to Unreal Engine[2], which leads to Figure (2). Finally, as for obtaining the waypoints from the building floor plan, we decided to achieve this interactively. We wrote a event callback function in UE4, such that when users right-click on the screen during gameplay, the game would automatically perform ray-tracing from the screen pixel location, and saves the location of the first-hit objects in the UE4 world. In addition, we also saved more information about the hit object (*i.e.,* its name, category, etc.) so that we could put into future use.

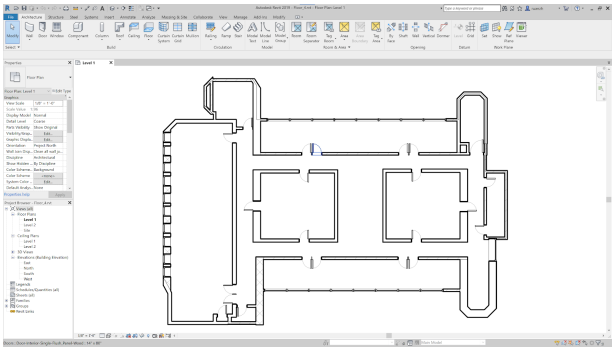

Fig. 1. The redrawn floor plan in Autodesk Revit

[1]https://www.unrealengine.com/en-US/datasmith/plugins
[2]https://docs.unrealengine.com/en-US/WorkingWithContent/Importing/Datasmith/SoftwareInteropGuides/Revit/index.html

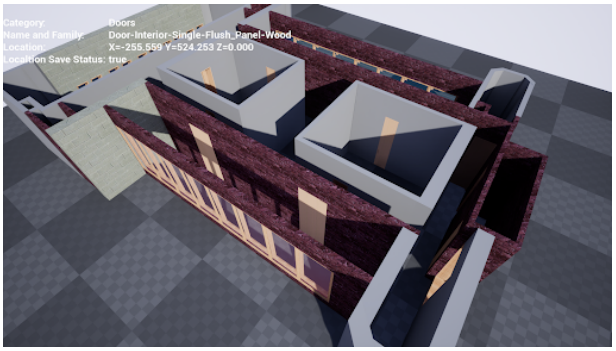

Fig. 2. The rendered floor plan in Unreal Engine 4.

B. Graph Planner and Waypoint Splitting

The planning stack of the project is built on the ROS framework. Given the waypoint locations in the floor plan in map space and k vehicles, the graph planner is designed to split waypoints into k groups and generate a path for each group such that the path visits each waypoint only once.

1) Clustering: An initial idea of the solution is to apply clustering algorithms directly in the map space, using the plain 2-D coordinates of the waypoints. However, as there are a large number of obstacles in the floor plan map and it is very likely that waypoints that seems together are actually very far away due to some existing wall in between, it is impractical to cluster them regardless of the locations of the walls. Therefore, it is important to construct a fully connected graph out of the waypoints so that we could apply penalties on the edges between two waypoints if there is an obstacle. By such means we would be able to depict the spatial relationship of the waypoints more accurately.

Spectral clustering [7] is good at clustering nodes in a graph based on the the edge weights. It first transforms the graph into graph Laplacian space using graph Laplacian transformation, and then apply another direct clustering algorithms in the Laplacian space. In our implementation, we constructed a complete graph from the input of waypoints, implemented the graph Laplacian transform on the graph, and then used k-means++ algorithm [8] as the direct clustering tool in the Laplacian space. As spectral clustering is based on the similarity of the nodes, we transformed the edge weights (which is initially distance-based) into similarity measures using Gaussian similarity metric. The result on the floor plan is shown in Figure (3).

After grouping the waypoints together, we still need to compute a close loop as the path for each group such that each path visits all points within its group exactly only once. This can be formulated as a Traveling Salesman Problem (TSP) and there have already existed a number of solvers to this problem. Therefore, given the clustering results, we first trim the edges in the original graph that connects points in different groups in order to generate k subgraphs. Next, we run a TSP solver[3] which is heavily inspired by [9] in each subgraph and get k TSP paths for our vehicles.

The downside of applying clustering algorithms to split the waypoints is that the equal-length constraint cannot be enforced in the generated paths, as there would be different number of waypoints in each clustered group. Figure (3) indicates such limitations, showing that there are fewer points in group 1 than group 2.

2) Capacitated Vehicle Routing Problem: We can also solve the problem by formulating it into a Capacitated Vehicle Routing Problem (CVRP). General VRP assumes that all vehicles start from the same place, and aims at computing equal-length paths in a graph such that all paths cover the entire set of graph nodes. As TSP solutions would be a trivial solution to an unconstrained VRP problem, we need to formulate a constrained VRP where each vehicle should only be able to visit a maximum number of nodes, which is also known as a CVRP. Ant Colony Optimization can be used to fast generate a reasonable approximation to the CVRP [10]. In our implementation, we applied the Ant Colony Optimization algorithm on the waypoint splitting problem and generated results on the floor plan as Figure (4) shows.

The limitation of applying CVRP solver to the waypoint splitting problem is that all vehicles are required to start from the same place. However, as our graph planner is very likely to re-plan as the F1/10 racing cars detect more obstacles using LiDAR, it is difficult for all cars to gather together once they are on the way to their own designated region.

3) Graph Partitioning: We can also formulate the problem into a graph partitioning problem. A graph partitioning problem aims at directly dividing the graph into k subgraphs such that the graph cut is minimized. We can run the TSP solver in the subgraphs as previously indicated to solve for TSP loops. This fits into our problem setting since if we manage to divide the original graph into k even components, then the TSP paths from subgraphs would also be more even. In our implementation, we applied multilevel k-way graph partitioning algorithm [3]. In order to produce the minimum overlapping subgraphs, we transformed the original edge weights into Gaussian similarity. By such means the algorithm is able to maximize the edge cut of our waypoint graph. Our implementation produced the generated TSP paths for each subgraph as shown in Figure (5).

Notice that of all the three methods, graph partitioning appears to have the best performance and the most flexibility. As a result, we decided to incorporate the graph partitioning algorithm into our graph planner.

[3]Please check Google OR-Tools for more information.

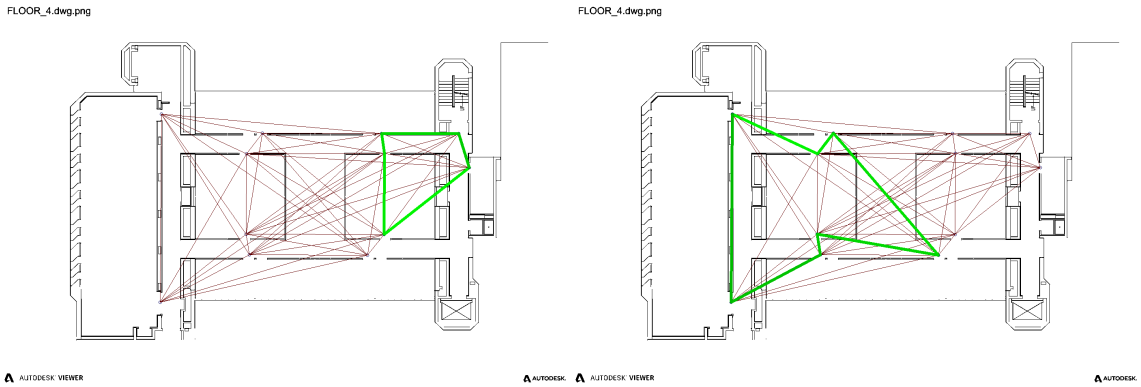

Fig. 3. Closed loop path for 2 vehicles using spectral clustering. Left: TSP path for Vehicle 1; right: TSP path for Vehicle 2.

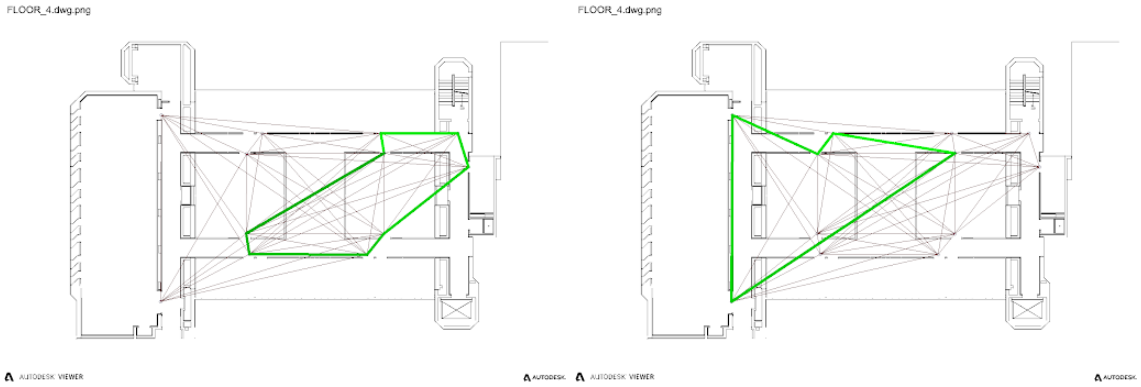

Fig. 4. Closed loop path for 2 vehicles by solving CVRP. Left: CVRP path for Vehicle 1; right: CVRP path for Vehicle 2.

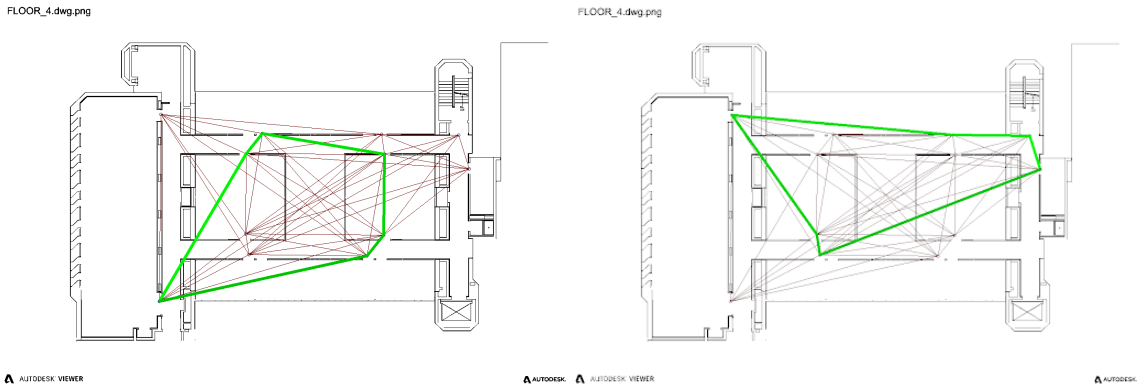

Fig. 5. Closed loop path for 2 vehicles by solving graph partitioning problem. Left: TSP path for Vehicle 1; right: TSP path for Vehicle 2.

C. Local Planner and Controller

Given the generated waypoint paths for k racing cars, we would like to develop a low-level planner and a controller to navigate and drive the cars in parallel. As the cars would be exploring an unknown environment, more obstacles are expected to be seen and our planner must be capable of dynamically avoid obstacles. In order to achieve this, we decided to apply FMT* [4] which is a sampling-based planning algorithm. In our implementation, we turned the local planner into a ROS action server, and queried the planning service in real time. We also keep tracking of the current position of each race car and update the goal point as the next waypoint in the local planner once the current goal is reached.

We used the "Pure Pursuit" algorithm [5] to design our controller for the race car. Specifically, given the next trajectory generated from our FMT* local planner, we interpolate an arc between the goal and our current position. Then, we set the steering angle of the vehicle proportional to the curvature of the arc. The desired speed of the vehicle is constant.

The implementation of the local planner and the controller relies heavily on the base code of this project, which was finished last year by Yash Trikannad and Saumya-Shah. The simulation of the race car navigation is performed on the F1TENTH Simulator[4] [1].

D. Perception and Mapping

In order to perceive the environment, we would like to incorporate Google Cartographer [6] into our system. This module is still under heavy development and more plans can be found in Section IV.

III. Implementation Details

The entire project is built in the ROS Noetic framework with C++ 17 on Ubuntu 20.04 LTS. In order for the code to compile, one may need the following dependencies:

- Boost 1.71
- LibConfig++ 1.5.0
- OpenCV 4.5.0
- Google OR-Tools 8.2
- OSQP 0.6.2
- OSQP-Eigen 0.6.3
- METIS 5.1.0

The UE4 rendering software is built with Unreal Engine 4.23.1 on Ubuntu 20.04 LTS.

A. Event Callback in UE4 Rendering

The waypoint saving module in the UE4 rendering software is designed to be an event callback loop subscribing to the mouse right-click activity. This was achieved through an Unreal Blueprint Design. The detailed Unreal Blueprint logic flow is indicated in Figure (6). As UE4 does not have a built-in Blueprint module for waypoing saving, a customized module is also implemented in C++.

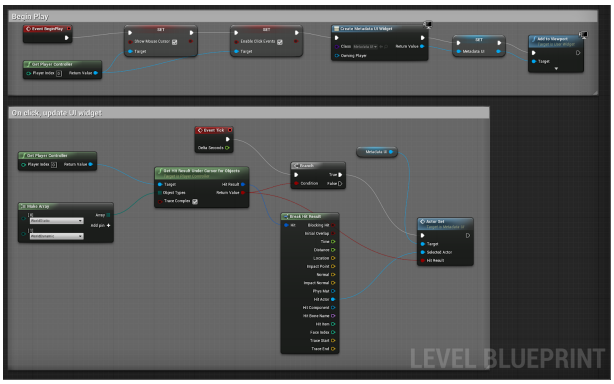

Fig. 6. Unreal Blueprint design for waypoint saving.

[4]https://f1tenth.readthedocs.io/en/stable/going_forward/simulator/index.html#

B. Spectral Clustering

Spectral clustering starts with the construction of the graph adjacency matrix. Given the waypoint graph, we first transformed the edge weights from waypoint distance to similarity measures using Gaussian similarity measure:

$$s_i = \exp\left(-\frac{e_i^2}{2\sigma^2}\right) \tag{1}$$

where σ is some parameter. In order to get a reasonable similarity value for each waypoint graph edge we set $\sigma = 0.5 \times \max\{e_1, e_2, \ldots, e_m\}$. Such measures essentially states that closer waypoints are considered as more "similar", which fits in our task of grouping closer waypoints together.

After constructing the adjacency matrix of the graph with Gaussian similarity, we then computed the graph Laplacian in Eigen C++. Graph Laplacian can be computed as

$$L = D - W = \mathbf{diag}\left(\begin{bmatrix} \sum_{j=1}^{m} W_{1j} \\ \vdots \\ \sum_{j=1}^{m} W_{mj} \end{bmatrix}\right) - W \tag{2}$$

where D is the degree matrix and W is the adjacency matrix. For grouping the graph nodes into k clusters, we then extracted the first k-smallest eigenvectors of the Laplacian matrix. Each row in the resulting matrix containing the k column vectors represents a feature point of the corresponding graph node (indexed from 1 to N) in the graph Laplacian space $\mathbb{R}^k$.

Our final task is to run a low-level clustering algorithm over the feature points in $\mathbb{R}^k$. The algorithm we chose was k-means++, which is an updated version of the original k-means algorithm with advanced cluster center initialization. A description of k-means++ algorithm is listed in Algorithm 1.

Algorithm 1: k-means++ Algorithm

Input : Data points $x_1, x_2, \ldots, x_n$
Output: Clustering centroids $c_1, c_2, \ldots, c_k$
while *number of initialized centroids is less than* k **do**
  Pick the farthest data point to the current centroids as a new centroid;
  Re-assign each data point to the closest centroid;
end
while *Cluster assignment does not converge* **do**
  Re-assign each data point to the closest centroid;
  Re-compute each centroid location as the mean of all current assigned data points;
end

C. Ant Colony Optimization for CVRP

We applied Ant Colony Optimization to solve the CVRP problem. Ant Colony Optimization is essentially an algorithm that mimics the behaviors of a group of ants when they are hunting for food. At first, each ant would randomly pick a direction to explore. They would leave some special kind of pheromones as they are walking. As long as an ant finds some food, it would notify other ants so that more and more ants would explore following its trail. Eventually the ants would find a path that leads them to the food.

Stodola et al. [10] proposes to use Ant Colony Optimization to solve the Capacitated Vehicle Routing Problem (CVRP). Our implementation of Ant Colony Optimization is based on the project base code and it can be summarized in Algorithm 2.

Algorithm 2: Ant Colony Optimization

```
for a number of iterations do
    for each ant in the colony do
        Set all nodes to unvisited;
        while number of unvisited nodes > 0 do
            compute ant's probability of going to
             next nodes;
            choose an unvisited node for the ant
             to go next;
            if ant's cost + distance to next node
             > ant's capacity then
                return to the starting point;
            end
            else
                visit the next node;
            end
        end
        return to the starting point;
    end
    save the solution with minimum cost;
    evaporate pheromone trails;
    update pheromone trails;
end
```

The pheromone trail is a matrix that records the performance of all ant's paths in each iteration, and its update involves increasing the "pheromone" along the best solution of that iteration (strengthen the best solution and increase probability for ants to explore in the next iteration) and decreasing the "pheromone" in other possible paths (pheromone evaporation).

Last but not least, given the best CVRP solution found in Ant Colony Optimization, we also ran a 2-Opt local search [11] to further improve the results.

D. Graph Partitioning

We applied the multilevel k-way graph partitioning algorithm [3] to solve the graph partitioning problem. The algorithm involves coarsening the graph node, form simple graphs, apply a simple graph bisection heuristic (such as Kernighan-Lin heuristic [12]) on the coarse graph, and then uncoarsen the graph to produce results. As Karypis et al. [3] also provides a nice software[5] encapsulating their graph partitioning ideas, we did not implement the algorithm from scratch. Instead, we managed to make use of the software directly for our project.

IV. Future Work

Our current system solves the problem of generating 3D model as well as waypoints from the floor plan of the building. It also addresses the solution of splitting the waypoints for k vehicles such that all vehicles can explore the building collaboratively and parallelly with equal effort and minimum overlap. However, it is still not able to perceive the environment in detail and update the 3D rendering scene in UE4. Some thoughts on the future plans of this project includes:

- Build a SLAM map of the environment using Google Cartographer;
- Use the SLAM map to update the position of obstacles in the graph planner, and possibly update the waypoint graphs and re-split the unvisited waypoints with graph partition;
- Perform point cloud smoothing and object detection on the LiDAR data, associate the LiDAR detection results with the objects in the 3D rendering scene. Identify doors, windows, walls, etc;
- Pass the point cloud data to UE4 gameplay, and update the 3D model with additional detailed information.

V. Conclusion

We have finished the 3D model setup and collaborative waypoint planning of this project. We have developed a complete pipeline from the floor plan of the building to rendering and multi-vehicle navigation in the floor plan. In particular, we used UE4 to render the model and generate the waypoints, and applied 3 different methods (spectral clustering, CVRP solving, graph partitioning) to the collaborative planning module of the project. Among the three proposed methods, graph partitioning was considered as the most promising results and it has been integrated to our planning stack. We applied a novel sampling-based algorithm to achieve the goal of obstacle avoidance in navigation and control of the race car. Future work of this project would be more focused on the perception module, which involves the use of Google Cartographer.

[5]See METIS: http://glaros.dtc.umn.edu/gkhome/metis/metis/overview

REFERENCES

[1] M. O'Kelly, H. Zheng, D. Karthik, and R. Mangharam, "F1tenth: An open-source evaluation environment for continuous control and reinforcement learning," in *Proceedings of the NeurIPS 2019 Competition and Demonstration Track*, ser. Proceedings of Machine Learning Research, H. J. Escalante and R. Hadsell, Eds., vol. 123. PMLR, 08–14 Dec 2020, pp. 77–89. [Online]. Available: http://proceedings.mlr.press/v123/o-kelly20a.html

[2] A. Agnihotri, M. O'Kelly, H. Abbas, and R. Mangharam, "Teaching autonomous systems at 1/10th-scale: A project-based course and community," in *ACM Special Interest Group on Computer Science Education (SIGCSE)*. ACM, 2020.

[3] G. Karypis and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," *Journal of Parallel and Distributed Computing*, vol. 48, no. 1, pp. 96–129, 1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731597914040

[4] L. Janson, E. Schmerling, A. Clark, and M. Pavone, "Fast marching tree: a fast marching sampling-based method for optimal motion planning in many dimensions," 2015.

[5] R. C. Coulter, "Implementation of the pure pursuit path tracking algorithm," Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-92-01, January 1992.

[6] W. Hess, D. Kohler, H. Rapp, and D. Andor, "Real-time loop closure in 2d lidar slam," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 2016, pp. 1271–1278.

[7] A. Y. Ng, M. I. Jordan, and Y. Weiss, "On spectral clustering: Analysis and an algorithm," in *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, ser. NIPS'01. Cambridge, MA, USA: MIT Press, 2001, p. 849–856.

[8] D. Arthur and S. Vassilvitskii, "K-means++: The advantages of careful seeding," in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '07. USA: Society for Industrial and Applied Mathematics, 2007, p. 1027–1035.

[9] P. Shaw, V. Furnon, and B. Backer, *A Constraint Programming Toolkit for Local Search*, 04 2006, vol. 18, pp. 219–261.

[10] P. Stodola, J. Mazal, M. Podhorec, and O. Litvaj, "Using the ant colony optimization algorithm for the capacitated vehicle routing problem," in *Proceedings of the 16th International Conference on Mechatronics - Mechatronika 2014*, 2014, pp. 503–510.

[11] G. A. Croes, "A method for solving traveling-salesman problems," *Operations Research*, vol. 6, no. 6, pp. 791–812, 1958. [Online]. Available: https://doi.org/10.1287/opre.6.6.791

[12] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970.